IEC 61508-7

Edition 2.0 2010-04

INTERNATIONAL STANDARD

NORME INTERNATIONALE

Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures

Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 7: Présentation de techniques et mesures

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub
The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, withdrawn and replaced publications.
- IEC Just Published: www.iec.ch/online_news/justpub
Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.
- Electropedia: www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.
- Customer Service Centre: www.iec.ch/webstore/custserv
If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:
Email: csc@iec.ch
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00

A propos de la CEI

La Commission Electrotechnique Internationale (CEI) est la première organisation mondiale qui élabore et publie des normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications CEI

Le contenu technique des publications de la CEI est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

- Catalogue des publications de la CEI: www.iec.ch/searchpub/cur_fut-f.htm
Le Catalogue en-ligne de la CEI vous permet d'effectuer des recherches en utilisant différents critères (numéro de référence, texte, comité d'études,…). Il donne aussi des informations sur les projets et les publications retirées ou remplacées.
- Just Published CEI: www.iec.ch/online_news/justpub
Restez informé sur les nouvelles publications de la CEI. Just Published détaille deux fois par mois les nouvelles publications parues. Disponible en-ligne et aussi par email.
- Electropedia: www.electropedia.org
Le premier dictionnaire en ligne au monde de termes électroniques et électriques. Il contient plus de 20 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans les langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International en ligne.
- Service Clients: www.iec.ch/webstore/custserv/custserv_entry-f.htm
Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions, visitez le FAQ du Service clients ou contactez-nous:
Email: csc@iec.ch
Tél.: +41 22 919 02 11
Fax: +41 22 919 03 00

IEC 61508-7

Edition 2.0 2010-04

INTERNATIONAL STANDARD

NORME INTERNATIONALE

Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures

Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 7: Présentation de techniques et mesures

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

PRICE CODE XG
CODE PRIX

ICS 25.040.40; 35.240.50

ISBN 978-2-88910-530-4

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

FUNCTIONAL SAFETY OF ELECTRICAL/ELECTRONIC/ PROGRAMMABLE ELECTRONIC SAFETY-RELATED SYSTEMS –

Part 7: Overview of techniques and measures

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61508-7 has been prepared by subcommittee 65A: System aspects, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2000. This edition constitutes a technical revision.

This edition has been subject to a thorough review and incorporates many comments received at the various revision stages.

The text of this standard is based on the following documents:

FDIS	Report on voting
65A/554/FDIS	65A/578/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts of the IEC 61508 series, published under the general title *Functional safety of electrical / electronic / programmable electronic safety-related systems*, can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

INTRODUCTION

Systems comprised of electrical and/or electronic elements have been used for many years to perform safety functions in most application sectors. Computer-based systems (generically referred to as programmable electronic systems) are being used in all application sectors to perform non-safety functions and, increasingly, to perform safety functions. If computer system technology is to be effectively and safely exploited, it is essential that those responsible for making decisions have sufficient guidance on the safety aspects on which to make these decisions.

This International Standard sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic (E/E/PE) elements that are used to perform safety functions. This unified approach has been adopted in order that a rational and consistent technical policy be developed for all electrically-based safety-related systems. A major objective is to facilitate the development of product and application sector international standards based on the IEC 61508 series.

NOTE 1 Examples of product and application sector international standards based on the IEC 61508 series are given in the bibliography (see references [21], [22] and [37]).

In most situations, safety is achieved by a number of systems which rely on many technologies (for example mechanical, hydraulic, pneumatic, electrical, electronic, programmable electronic). Any safety strategy must therefore consider not only all the elements within an individual system (for example sensors, controlling devices and actuators) but also all the safety-related systems making up the total combination of safety-related systems. Therefore, while this International Standard is concerned with E/E/PE safety-related systems, it may also provide a framework within which safety-related systems based on other technologies may be considered.

It is recognized that there is a great variety of applications using E/E/PE safety-related systems in a variety of application sectors and covering a wide range of complexity, hazard and risk potentials. In any particular application, the required safety measures will be dependent on many factors specific to the application. This International Standard, by being generic, will enable such measures to be formulated in future product and application sector international standards and in revisions of those that already exist.

This International Standard

– considers all relevant overall, E/E/PE system and software safety lifecycle phases (for example, from initial concept, through design, implementation, operation and maintenance to decommissioning) when E/E/PE systems are used to perform safety functions;
– has been conceived with a rapidly developing technology in mind; the framework is sufficiently robust and comprehensive to cater for future developments;
– enables product and application sector international standards, dealing with E/E/PE safety-related systems, to be developed; the development of product and application sector international standards, within the framework of this standard, should lead to a high level of consistency (for example, of underlying principles, terminology etc.) both within application sectors and across application sectors; this will have both safety and economic benefits;
– provides a method for the development of the safety requirements specification necessary to achieve the required functional safety for E/E/PE safety-related systems;
– adopts a risk-based approach by which the safety integrity requirements can be determined;
– introduces safety integrity levels for specifying the target level of safety integrity for the safety functions to be implemented by the E/E/PE safety-related systems;

NOTE 2 The standard does not specify the safety integrity level requirements for any safety function, nor does it mandate how the safety integrity level is determined. Instead it provides a risk-based conceptual framework and example techniques.

– sets target failure measures for safety functions carried out by E/E/PE safety-related systems, which are linked to the safety integrity levels;

– sets a lower limit on the target failure measures for a safety function carried out by a single E/E/PE safety-related system. For E/E/PE safety-related systems operating in

  – a low demand mode of operation, the lower limit is set at an average probability of a dangerous failure on demand of 10^{-5};

  – a high demand or a continuous mode of operation, the lower limit is set at an average frequency of a dangerous failure of 10^{-9} [h^{-1}];

NOTE 3 A single E/E/PE safety-related system does not necessarily mean a single-channel architecture.

NOTE 4 It may be possible to achieve designs of safety-related systems with lower values for the target safety integrity for non-complex systems, but these limits are considered to represent what can be achieved for relatively complex systems (for example programmable electronic safety-related systems) at the present time.

– sets requirements for the avoidance and control of systematic faults, which are based on experience and judgement from practical experience gained in industry. Even though the probability of occurrence of systematic failures cannot in general be quantified the standard does, however, allow a claim to be made, for a specified safety function, that the target failure measure associated with the safety function can be considered to be achieved if all the requirements in the standard have been met;

– introduces systematic capability which applies to an element with respect to its confidence that the systematic safety integrity meets the requirements of the specified safety integrity level;

– adopts a broad range of principles, techniques and measures to achieve functional safety for E/E/PE safety-related systems, but does not explicitly use the concept of fail safe. However, the concepts of “fail safe” and “inherently safe” principles may be applicable and adoption of such concepts is acceptable providing the requirements of the relevant clauses in the standard are met.

FUNCTIONAL SAFETY OF ELECTRICAL/ELECTRONIC/ PROGRAMMABLE ELECTRONIC SAFETY-RELATED SYSTEMS –

Part 7: Overview of techniques and measures

1 Scope

1.1 This part of IEC 61508 contains an overview of various safety techniques and measures relevant to IEC 61508-2 and IEC 61508-3.

The references should be considered as basic references to methods and tools or as examples, and may not represent the state of the art.

1.2 IEC 61508-1, IEC 61598-2, IEC 61508-3 and IEC 61508-4 are basic safety publications, although this status does not apply in the context of low complexity E/E/PE safety-related systems (see 3.4.3 of IEC 61508-4). As basic safety publications, they are intended for use by technical committees in the preparation of standards in accordance with the principles contained in IEC Guide 104 and ISO/IEC Guide 51. IEC 61508-1, IEC 61508-2, IEC 61508-3 and IEC 61508-4 are also intended for use as stand-alone publications. The horizontal safety function of this international standard does not apply to medical equipment in compliance with the IEC 60601 series.

1.3 One of the responsibilities of a technical committee is, wherever applicable, to make use of basic safety publications in the preparation of its publications. In this context, the requirements, test methods or test conditions of this basic safety publication will not apply unless specifically referred to or included in the publications prepared by those technical committees.

1.4 Figure 1 shows the overall framework for parts 1 to 7 of IEC 61508 and indicates the role that IEC 61508-7 plays in the achievement of functional safety for E/E/PE safety-related systems.

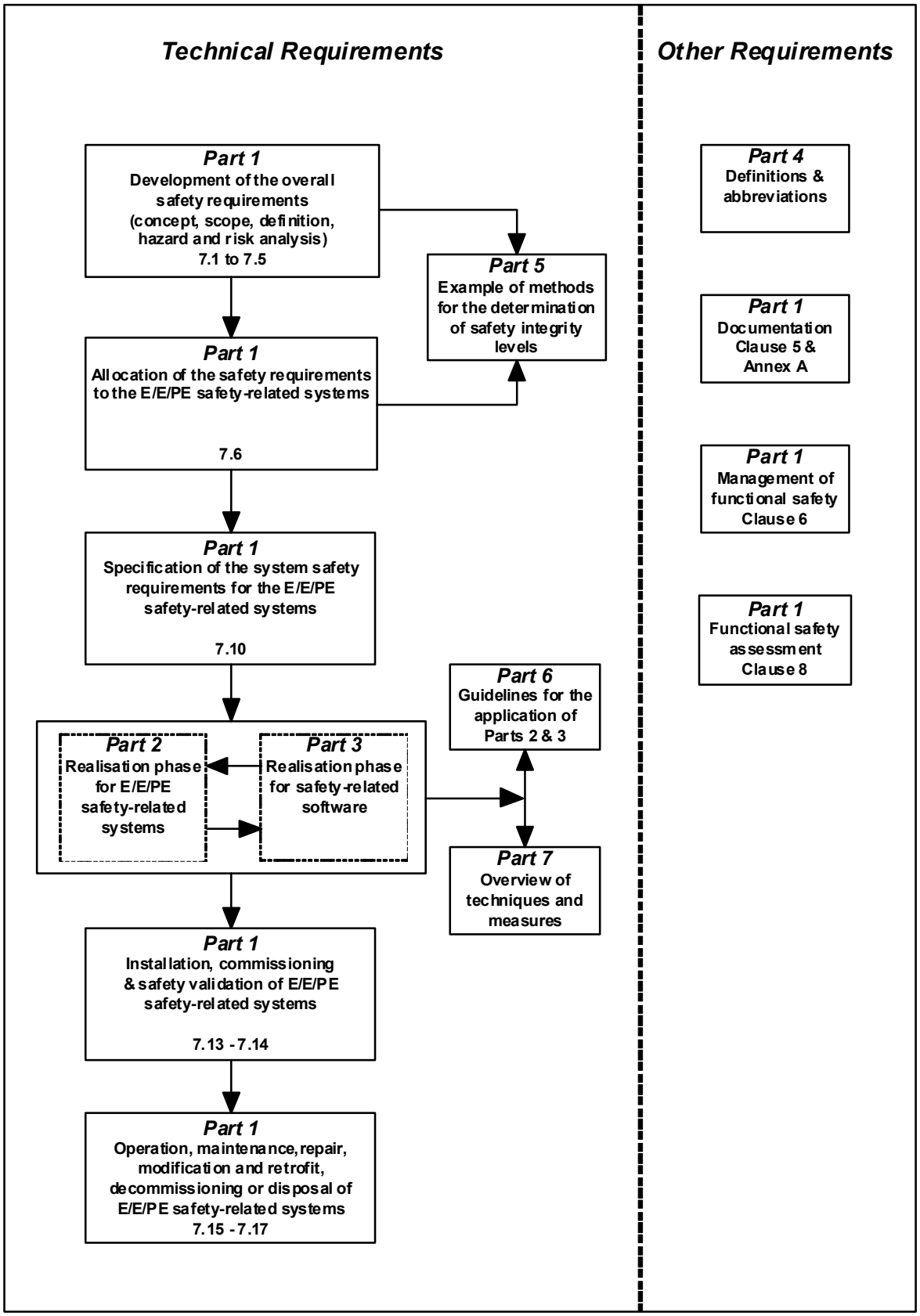


Figure 1 – Overall framework of IEC 61508

2 Normative references

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

IEC 61508-4:2010 *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 4: Definitions and abbreviations*

3 Definitions and abbreviations

For the purposes of this document, the definitions and abbreviations given in IEC 61508-4 apply.

Annex A
(informative)

Overview of techniques and measures for E/E/PE safety-related systems: control of random hardware failures (see IEC 61508-2)

A.1 Electric

Global objective: To control failures in electromechanical components.

A.1.1 Failure detection by on-line monitoring

NOTE This technique/measure is referenced in Tables A.2, A.3, A.7 and A.13 to A.18 of IEC 61508-2.

Aim: To detect failures by monitoring the behaviour of the E/E/PE safety-related system in response to the normal (on-line) operation of the equipment under control (EUC).

Description: Under certain conditions, failures can be detected using information about (for example) the time behaviour of the EUC. For example, if a switch, which is part of the E/E/PE safety-related system, is normally actuated by the EUC, then if the switch does not change state at the expected time, a failure will have been detected. It is not usually possible to localise the failure.

A.1.2 Monitoring of relay contacts

NOTE This technique/measure is referenced in Tables A.2 and A.14 of IEC 61508-2.

Aim: To detect failures (for example welding) of relay contacts.

Description: Forced contact (or positively guided contact) relays are designed so that their contacts are rigidly linked together. Assuming there are two sets of changeover contacts, *a* and *b*, if the normally open contact, *a*, welds, the normally closed contact, *b*, cannot close when the relay coil is next de-energised. Therefore, the monitoring of the closure of the normally closed contact *b* when the relay coil is de-energised may be used to prove that the normally open contact *a* has opened. Failure of normally closed contact *b* to close indicates a failure of contact *a*, so the monitoring circuit should ensure a safe shut-down, or ensure that shut-down is continued, for any machinery controlled by contact *a*.

References:

Zusammenstellung und Bewertung elektromechanischer Sicherheitsschaltungen für Verriegelungseinrichtungen. F. Kreutzkampf, W. Hertel, Sicherheitstechnisches Informations- und Arbeitsblatt 330212, BIA-Handbuch. 17. Lfg. X/91, Erich Schmidt Verlag, Bielefeld.
www.BGIA-HANDBUCHdigital.de/330212

A.1.3 Comparator

NOTE This technique/measure is referenced in Tables A.2, A.3, A.4 of IEC 61508-2.

Aim: To detect, as early as possible, (non-simultaneous) failures in an independent processing unit or in the comparator.

Description: The signals of independent processing units are compared cyclically or continuously by a hardware comparator. The comparator may itself be externally tested, or it may use self-monitoring technology. Detected differences in the behaviour of the processors lead to a failure message.

A.1.4 Majority voter

NOTE This technique/measure is referenced in Tables A.2, A.3 and A.4 of IEC 61508-2.

Aim: To detect and mask failures in one of at least three hardware channels.

Description: A voting unit using the majority principle (2 out of 3, 3 out of 3, or *m* out of *n*) is used to detect and mask failures. The voter may itself be externally tested, or it may use self-monitoring technology.

References:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

A.1.5 Idle current principle (de-energised to trip)

NOTE This technique/measure is referenced in Table A.16 of IEC 61508-2.

Aim: To execute the safety function if power is cut or lost.

Description: The safety function is executed if the contacts are open and no current flows. For example, if brakes are used to stop a dangerous movement of a motor, the brakes are opened by closing contacts in the safety-related system and are closed by opening the contacts in the safety-related system.

Reference:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

A.2 Electronic

Global objective: To control failure in solid-state components.

A.2.1 Tests by redundant hardware

NOTE This technique/measure is referenced in Tables A.3, A.15, A.16 and A.18 of IEC 61508-2.

Aim: To detect failures using hardware redundancy, i.e. using additional hardware not required to implement the process functions.

Description: Redundant hardware can be used to test at an appropriate frequency the specified safety functions. This approach is normally necessary for realising A.1.1 or A.2.2.

A.2.2 Dynamic principles

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-2.

Aim: To detect static failures by dynamic signal processing.

Description: A forced change of otherwise static signals (internally or externally generated) helps to detect static failures in components. This technique is often associated with electromechanical components.

Reference:

Elektronik in der Sicherheitstechnik. H. Jürs, D. Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch, Erich-Schmidt Verlag, Bielefeld, 1993. http://www.bgia-handbuchdigital.de/330220

A.2.3 Standard test access port and boundary-scan architecture

NOTE This technique/measure is referenced in Tables A.3, A.15 and A.18 of IEC 61508-2.

Aim: To control and observe what happens at each pin of an IC.

Description: Boundary-scan test is an IC design technique which increases the testability of the IC by resolving the problem of how to gain access to the circuit test points within it. In a typical boundary-scan IC, comprised of core logic and input and output buffers, a shift-register stage is placed between the core logic and the input and output buffers adjacent to each IC pin. Each shift-register stage is contained in a boundary-scan cell. The boundary-scan cell can control and observe what happens at each input and output pin of an IC, via the standard test access port. Internal testing of the IC core logic is accomplished by isolating the on-chip core logic from stimuli received from surrounding components, and then performing an internal self-test. These tests can be used to detect failures in the IC.

Reference:

IEEE 1149-1:2001, *IEEE standard test access port and boundary-scan architecture, IEEE Computer Society*, 2001, ISBN: 0-7381-2944-5

A.2.4 (Not used)

A.2.5 Monitored redundancy

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-2.

Aim: To detect failure, by providing several functional units, by monitoring the behaviour of each of these to detect failures, and by initiating a transition to a safe condition if any discrepancy in behaviour is detected.

Description: The safety function is executed by at least two hardware channels. The outputs of these channels are monitored and a safe condition is initiated if a fault is detected (i.e. if the output signals from all channels are not identical).

References:

Elektronik in der Sicherheitstechnik. H. Jürs, D. Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch, Erich-Schmidt Verlag, Bielefeld, 1993. http://www.bgia-handbuchdigital.de/330220

Dependability of Critical Computer Systems 1. F. J. Redmill, Elsevier Applied Science, 1988, ISBN 1-85166-203-0

A.2.6 Electrical/electronic components with automatic check

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-2.

Aim: To detect faults by periodic checking of the safety functions.

Description: The hardware is tested before starting the process, and is tested repeatedly at suitable intervals. The EUC continues to operate only if each test is successful.

References:

Elektronik in der Sicherheitstechnik. H. Jürs, D. Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch, Erich-Schmidt Verlag, Bielefeld, 1993. http://www.bgia-handbuchdigital.de/330220

Dependability of Critical Computer Systems 1. F. J. Redmill, Elsevier Applied Science, 1988, ISBN 1-85166-203-0

A.2.7 Analogue signal monitoring

NOTE This technique/measure is referenced in Tables A.3 and A.13 of IEC 61508-2.

Aim: To improve confidence in measured signals.

Description: Wherever there is a choice, analogue signals are used in preference to digital on/off states. For example, trip or safe states are represented by analogue signal levels, usually with signal level tolerance monitoring. The technique provides continuity monitoring and a higher level of confidence in the transmitter, reducing the necessary proof-test frequency of the transmitter sensing function. External interfaces, for example impulse lines, will also require testing.

A.2.8 De-rating

NOTE This technique/measure is referenced in 7.4.2.13 of IEC 61508-2.

Aim: To increase the reliability of hardware components.

Description: Hardware components are operated at levels which are guaranteed by the design of the system to be well below the maximum specification ratings. De-rating is the practice of ensuring that under all normal operating circumstances, components are operated well below their maximum stress levels.

A.3 Processing units

Global objective: To recognise failures which lead to incorrect results in processing units.

A.3.1 Self-test by software: limited number of patterns (one-channel)

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-2.

Aim: To detect, as early as possible, failures in the processing unit.

Description: The hardware is built using standard techniques which do not take any special safety requirements into account. The failure detection is realised entirely by additional software functions which perform self-tests using at least two complementary data patterns (for example 55hex and AAhex).

A.3.2 Self-test by software: walking bit (one-channel)

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-2.

Aim: To detect, as early as possible, failures in the physical storage (for example registers) and instruction decoder of the processing unit.

Description: The failure detection is realised entirely by additional software functions which perform self-tests using a data pattern (for example walking-bit pattern) which tests the physical storage (data and address registers) and the instruction decoder. However, the diagnostic coverage is only 90 %.

A.3.3 Self-test supported by hardware (one-channel)

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-2.

Aim: To detect, as early as possible, failures in the processing unit, using special hardware that increases the speed and extends the scope of failure detection.

Description: Additional special hardware facilities support self-test functions to detect failure. For example, this could be a hardware unit which cyclically monitors the output of a certain bit pattern according to the watch-dog principle.

A.3.4 Coded processing (one-channel)

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-2.

Aim: To detect, as early as possible, failures in the processing unit.

Description: Processing units can be designed with special failure-recognising or failure-correcting circuit techniques. So far, these techniques have been applied only to relatively simple circuits and are not widespread; however, future developments should not be excluded.

References:

Le processeur codé: un nouveau concept appliqué à la sécurité des systèmes de transports. Gabriel, Martin, Wartski, Revue Générale des chemins de fer, No. 6, June 1990

Vital Coded Microprocessor Principles and Application for Various Transit Systems. P. Forin, IFAC Control Computers Communications in Transportation, 79-84, 1989

A.3.5 Reciprocal comparison by software

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-2.

Aim: To detect, as early as possible, failures in the processing unit, by dynamic software comparison.

Description: Two processing units exchange data (including results, intermediate results and test data) reciprocally. A comparison of the data is carried out using software in each unit and detected differences lead to a failure message.

A.4 Invariable memory ranges

Global objective: The detection of information modifications in the invariable memory.

A.4.1 Word-saving multi-bit redundancy (for example ROM monitoring with a modified Hamming code)

NOTE 1 This technique/measure is referenced in Table A.5 of IEC 61508-2.

NOTE 2 See also A.5.6 “RAM monitoring with a modified Hamming code, or detection of data failures with error-detection-correction codes (EDC)” and C.3.2 “Error detecting and correcting codes”.

Aim: To detect all single-bit failures, all two-bit failures, some three-bit failures, and some all-bit failures in a 16-bit word.

Description: Every word of memory is extended by several redundant bits to produce a modified Hamming code with a Hamming distance of at least 4. Every time a word is read, checking of the redundant bits can determine whether or not a corruption has taken place. If a difference is found, a failure message is produced. The procedure can also be used to detect

addressing failures, by calculating the redundant bits for the concatenation of the data word and its address.

References:

Prüfbare und korrigierbare Codes. W. W. Peterson, München, Oldenburg, 1967

Error detecting and error correcting codes. R. W. Hamming, The Bell System Technical Journal 29 (2), 147-160, 1950

A.4.2 Modified checksum

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-2.

Aim: To detect all odd-bit failures, i.e. approximately 50 % of all possible bit failures.

Description: A checksum is created by a suitable algorithm which uses all the words in a block of memory. The checksum may be stored as an additional word in ROM, or an additional word may be added to the memory block to ensure that the checksum algorithm produces a predetermined value. In a later memory test, a checksum is created again using the same algorithm, and the result is compared with the stored or defined value. If a difference is found, a failure message is produced.

A.4.3 Signature of one word (8-bit)

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-2.

Aim: To detect all one-bit failures and all multi-bit failures within a word, as well as approximately 99,6 % of all possible bit failures.

Description: The contents of a memory block is compressed (using either hardware or software) using a cyclic redundancy check (CRC) algorithm into one memory word. A typical CRC algorithm treats the whole contents of the block as byte-serial or bit-serial data flow, on which a continued polynomial division is carried out using a polynomial generator. The remainder of the division represents the compressed memory contents – it is the "signature" of the memory – and is stored. The signature is computed once again in later tests and compared with one already stored. A failure message is produced if there is a difference.

A.4.4 Signature of a double word (16-bit)

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-2.

Aim: To detect all one-bit failures and all multi-bit failures within a word, as well as approximately 99,998 % of all possible bit failures.

Description: This procedure calculates a signature using a cyclic redundancy check (CRC) algorithm, but the resulting value is at least two words in size. The extended signature is stored, recalculated and compared as in the single-word case. A failure message is produced if there is a difference between the stored and recalculated signatures.

A.4.5 Block replication (for example double ROM with hardware or software comparison)

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-2.

Aim: To detect all bit failures.

Description: The address space is duplicated in two memories. The first memory is operated in the normal manner. The second memory contains the same information and is accessed in parallel to the first. The outputs are compared and a failure message is produced if a

difference is detected. In order to detect certain kinds of bit errors, the data must be stored inversely in one of the two memories and inverted once again when read.

A.5 Variable memory ranges

Global objective: Detecting failures during addressing, writing, storing and reading.

NOTE Soft-errors are listed in Table A.1, IEC 61508-2 as faults to be detected during operation or to be analysed in the derivation of the safe failure fraction. Causes of soft errors are: (1) Alpha particles from package decay, (2) Neutrons, (3) external EMI noise, (4) Internal cross-talk. External EMI noise is covered by other requirements of this international standard.

The effect of Alpha particles and Neutrons should be mastered by safety integrity measures at runtime. Safety integrity measures effective for hard errors may not be effective for soft errors, e.g. RAM tests, such as walk-path, galpat, etc. are not effective, whereas monitoring techniques such as Parity and ECC with recurring read of the memory cells are.

A soft error occurs when a radiation event causes enough of a charge disturbance to reverse or flip the data state of a low energized semiconductor memory cell, register, latch, or flip-flop. The error is called “soft” because the circuit itself is not permanently damaged by the radiation. Soft-errors are classified in Single Bit Upsets (SBU) or Single Event Upsets (SEU) and Multi-Bit Upsets (MBU).

If the disturbed circuit is a storage element like memory cell or flip-flop, the state is stored until the next (intended) write operation. The new data will be stored correctly. In a combinatory circuit the effect is rather a glitch because there is a continuous energy flow from the component driving this node. On connecting wires and communication lines the effect could also be a glitch. However due to the larger capacitance the effect by Alpha particles and Neutrons is considered negligible.

Soft-errors may be relevant to variable memory of any kind, i.e., to DRAM, SRAM, register banks in µP, cache, pipelines, configuration registers of devices such as ADC, DMA, MMU, Interrupt controller, complex timers. Sensitivity to alpha and neutron particles is a function of both core voltage and geometry. Smaller geometries at 2,5 V core voltage and especially below 1,8 V would require more evaluation and more effective protective measures.

The soft error rate has been reported (see a) and i) below) to be in a range of 700 Fit/MBit to 1 200 Fit/MBit for (embedded) memories. This is a reference value to be compared with data coming from the silicon process with which the device is implemented. Until recently SBU were considered to be dominant, but the latest forecast (see a) below) reports a growing percentage of MBU of the overall soft-error rate (SER) for technologies from 65 nm down.

The following literature and sources give details about soft-errors:

a) Altitude SEE Test European Platform (ASTEP) and First Results in CMOS 130 nm SRAM. J-L. Autran, P. Roche, C. Sudre et al. Nuclear Science, IEEE Transactions on Volume 54, Issue 4, Aug. 2007 Page(s):1002 - 1009

b) Radiation-Induced Soft Errors in Advanced Semiconductor Technologies, Robert C. Baumann, Fellow, IEEE, IEEE TRANSACTIONS ON DEVICE AND MATERIALS RELIABILITY, VOL. 5, NO. 3, SEPTEMBER 2005

c) Soft errors' impact on system reliability, Ritesh Mastipuram and Edwin C Wee, Cypress Semiconductor, 2004

d) Trends And Challenges In VLSI Circuit Reliability, C. Costantinescu, Intel, 2003, IEEE Computer Society

e) Basic mechanisms and modeling of single-event upset in digital microelectronics, P. E. Dodd and L. W. Massengill, IEEE Trans. Nucl. Sci., vol. 50, no. 3, pp. 583–602, Jun. 2003.

f) Destructive single-event effects in semiconductor devices and ICs, F. W. Sexton, IEEE Trans. Nucl. Sci., vol. 50, no. 3, pp. 603–621, Jun. 2003.

g) Coming Challenges in Microarchitecture and Architecture, Ronen, Mendelson, Proceedings of the IEEE, Volume 89, Issue 3, Mar 2001 Page(s):325 – 340

h) Scaling and Technology Issues for Soft Error Rates, A Johnston, 4th Annual Research Conference on Reliability Stanford University, October 2000

i) International Technology Roadmap for Semiconductors (ITRS), several papers.

A.5.1 RAM test "checkerboard" or "march"

NOTE This technique/measure is referenced in Table A.6 of IEC 61508-2.

Aim: To detect predominantly static bit failures.

Description: A checker-board type pattern of 0 s and 1 s is written into the cells of a bit-oriented memory. The cells are then inspected in pairs to ensure that the contents are the same and correct. The address of the first cell of such a pair is variable and the address of the second cell of the pair is formed by inverting bitwise the first address. In the first run, the address range of the memory is run towards higher addresses from the variable address, and in a second run towards lower addresses. Both runs are then repeated with an inverted pre-assignment. A failure message is produced if any difference occurs.

In a RAM test "march", the cells of a bit-oriented memory are initialised by a uniform bit stream. In the first run, the cells are inspected in ascending order: each cell is checked for the correct contents and its contents are inverted. The background, which is created in the first run, is treated in a second run in descending order and in the same manner. Both first runs are repeated with an inverted pre-assignment in a third or fourth run. A failure message is produced if a difference occurs.

A.5.2 RAM test "walkpath"

NOTE This technique/measure is referenced in Table A.6 of IEC 61508-2.

Aim: To detect static and dynamic bit failures and cross-talk between memory cells.

Description: The memory range to be tested is initialised by a uniform bit stream. The first cell is then inverted and the remaining memory area is inspected to ensure that the background is correct. After this, the first cell is re-inverted to return it to its original value, and the whole procedure is repeated for the next cell. A second run of the "wandering bit model" is carried out with an inverse background pre-assignment. A failure message is produced if a difference occurs.

A.5.3 RAM test "galpat" or "transparent galpat"

NOTE This technique/measure is referenced in Table A.6 of IEC 61508-2.

Aim: To detect static bit failures and a large proportion of dynamic couplings.

Description: In the RAM test "galpat", the chosen range of memory is first initialised uniformly (i.e. all 0 s or all 1 s). The first memory cell to be tested is then inverted and all the remaining cells are inspected to ensure that their contents are correct. After every read access to one of the remaining cells, the inverted cell is also checked. This procedure is repeated for each cell in the chosen memory range. A second run is carried out with the opposite initialisation. Any difference produces a failure message.

The "transparent galpat" test is a variation on the above procedure: instead of initialising all cells in the chosen memory range, the existing contents are left unchanged and signatures are used to compare the contents of sets of cells. The first cell to be tested in the chosen range is selected, and the signature S1 of all remaining cells in the range is calculated and stored. The cell to be tested is then inverted and the signature S2 of all the remaining cells is recalculated. (After every read access to one of the remaining cells, the inverted cell is also checked.) S2 is compared with S1, and any difference produces a failure message. The cell under test is re-inverted to re-establish the original contents, and the signature S3 of all the remaining cells is recalculated and compared with S1. Any difference produces a failure message. All memory cells in the chosen range are tested in the same manner.

A.5.4 RAM test "Abraham"

NOTE This technique/measure is referenced in Table A.6 of IEC 61508-2.

Aim: To detect all stuck-at and coupling failures between memory cells.

Description: The proportion of faults detected exceeds that of the RAM test "galpat". The number of operations required to perform the entire memory test is about 30 n, where n is the

number of cells in the memory. The test can be made transparent for use during the operating cycle by partitioning the memory and testing each partition in different time segments.

Reference:

Efficient Algorithms for Testing Semiconductor Random-Access Memories. R. Nair, S. M. Thatte, J. A. Abraham, IEEE Trans. Comput. C-27 (6), 572-576, 1978

A.5.5 One-bit redundancy (for example RAM monitoring with a parity bit)

NOTE This technique/measure is referenced in Table A.6 of IEC 61508-2.

Aim: To detect 50 % of all possible bit failures in the memory range tested.

Description: Every word of the memory is extended by one bit (the parity bit) which completes each word to an even or odd number of logical 1 s. The parity of the data word is checked each time it is read. If the wrong number of 1 s is found, a failure message is produced. The choice of even or odd parity should be made such that, whichever of the zero word (nothing but 0 s) and the one word (nothing but 1 s) is the more unfavourable in the event of a failure, then that word is not a valid code. Parity can also be used to detect addressing failures, when the parity is calculated for the concatenation of the data word and its address.

A.5.6 RAM monitoring with a modified Hamming code, or detection of data failures with error-detection-correction codes (EDC)

NOTE 1 This technique/measure is referenced in Table A.6 of IEC 61508-2.

NOTE 2 See also A.4.1 “Word-saving multi-bit redundancy (for example ROM monitoring with a modified Hamming code)”and C.3.2 “Error detecting and correcting codes”.

Aim: To detect all odd-bit failures, all two-bit failures, some three-bit and some multi-bit failures.

Description: Every access to memory is extended by several redundant bits to produce a modified Hamming code with a Hamming distance of at least 4. Every time data is read, one can determine whether a corruption has taken place by checking the redundant bits. If a difference is found, a failure message is produced. The procedure can also be used to detect addressing failure, when the redundant bits are calculated for the concatenation of the data and its address.

References:

Prüfbare und korrigierbare Codes. W. W. Peterson, München, Oldenburg, 1967

Error detecting and error correcting codes. R. W. Hamming, The Bell System Technical Journal 29 (2), 147-160, 1950

A.5.7 Double RAM with hardware or software comparison and read/write test

NOTE This technique/measure is referenced in Table A.6 of IEC 61508-2.

Aim: To detect all bit failures.

Description: The address space is duplicated in two memories. The first memory is operated in the normal manner. The second memory contains the same information and is accessed in parallel to the first. The outputs are compared and a failure message is produced if a difference is detected. In order to detect certain kinds of bit errors, the data must be stored inversely in one of the two memories and inverted once again when read.

A.6 I/O-units and interfaces (external communication)

Global objective: To detect failures in input and output units (digital, analogue, serial or parallel) and to prevent the sending of inadmissible outputs to the process.

A.6.1 Test pattern

NOTE This technique/measure is referenced in Tables A.7, A.13 and A.14 of IEC 61508-2.

Aim: To detect static failures (stuck-at failures) and cross-talk.

Description: This is a dataflow-independent cyclical test of input and output units. It uses a defined test pattern to compare observations with the corresponding expected values. The test pattern information, the test pattern reception, and test pattern evaluation must all be independent of each other. The EUC should not be inadmissibly influenced by the test pattern.

A.6.2 Code protection

NOTE This technique/measure is referenced in Tables A.7, A.15, A.16 and A.18 of IEC 61508-2.

Aim: To detect random hardware and systematic failures in the input/output dataflow.

Description: This procedure protects the input and output information from both systematic and random hardware failures. Code protection provides dataflow-dependent failure detection of the input and output units, based on information redundancy and/or time redundancy. Typically, redundant information is superimposed on input and/or output data. This then provides a means to monitor the correct operation of the input or output circuits. Many techniques are possible, for example a carrier frequency signal may be superimposed on the output signal of a sensor. The logic unit may then check for the presence of the carrier frequency or redundant code bits may be added to an output channel to allow the monitoring of the validity of a signal passing between the logic unit and final actuator.

A.6.3 Multi-channel parallel output

NOTE This technique/measure is referenced in Table A.7 of IEC 61508-2.

Aim: To detect random hardware failures (stuck-at failures), failures caused by external influences, timing failures, addressing failures, drift failures and transient failures.

Description: This is a dataflow-dependent multi-channel parallel output with independent outputs for the detection of random hardware failures. Failure detection is carried out via external comparators. If a failure occurs, the EUC is switched off directly. This measure is only effective if the dataflow changes during the diagnostic test interval.

A.6.4 Monitored outputs

NOTE This technique/measure is referenced in Table A.7 of IEC 61508-2.

Aim: To detect individual failures, failures caused by external influences, timing failures, addressing failures, drift failures (for analogue signals) and transient failures.

Description: This is a dataflow-dependent comparison of outputs with independent inputs to ensure compliance with a defined tolerance range (time, value). A detected failure cannot always be related to the defective output. This measure is only effective if the dataflow changes during the diagnostic test interval.

A.6.5 Input comparison/voting

NOTE This technique/measure is referenced in Tables A.7 and A.13 of IEC 61508-2.

Aim: To detect individual failures, failures caused by external influences, timing failures, addressing failures, drift failures (for analogue signals) and transient failures.

Description: This is a dataflow-dependent comparison of independent inputs to ensure compliance with a defined tolerance range (time, value). There will be 1 out of 2, 2 out of 3 or better redundancy. This measure is only effective if the dataflow changes during the diagnostic test interval.

A.7 Data paths (internal communication)

Global objective: To detect failures caused by a defect in the information transfer.

A.7.1 One-bit hardware redundancy

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-2.

Aim: To detect all odd-bit failures, i.e. 50 % of all the possible bit failures in the data stream.

Description: The bus is extended by one line (bit) and this additional line (bit) is used to detect failures by parity checking.

A.7.2 Multi-bit hardware redundancy

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-2.

Aim: To detect failures during the communication on the bus and in serial transmission links.

Description: The bus is extended by two or more lines (bits) and these additional lines (bits) are used in order to detect failures by Hamming code techniques.

A.7.3 Complete hardware redundancy

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-2.

Aim: To detect failures during the communication by comparing the signals on two buses.

Description: The bus is doubled and the additional lines (bits) are used in order to detect failures.

A.7.4 Inspection using test patterns

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-2.

Aim: To detect static failures (stuck-at failure) and cross-talk.

Description: This is a dataflow-independent cyclical test of data paths. It uses a defined test pattern to compare observations with the corresponding expected values.

The test pattern information, the test pattern reception, and test pattern evaluation must all be independent of each other. The EUC should not be inadmissibly influenced by the test pattern.

A.7.5 Transmission redundancy

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-2.

Aim: To detect transient failures in bus communication.

Description: The information is transferred several times in sequence. The repetition is effective only against transient failures.

A.7.6 Information redundancy

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-2.

Aim: To detect failures in bus communication.

Description: Data is transmitted in blocks, together with a calculated checksum for each block. The receiver then re-calculates the checksum of the received data and compares the result with the received checksum.

A.8 Power supply

Global objective: To detect or tolerate failures caused by a defect in the power supply.

A.8.1 Overvoltage protection with safety shut-off

NOTE This technique/measure is referenced in Table A.9 of IEC 61508-2.

Aim: To protect the safety-related system against overvoltage.

Description: Overvoltage is detected early enough that all outputs can be switched to a safe condition by the power-down routine or there is a switch-over to a second power unit.

Reference:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

A.8.2 Voltage control (secondary)

NOTE This technique/measure is referenced in Table A.9 of IEC 61508-2.

Aim: To monitor the secondary voltages and initiate a safe condition if the voltage is not in its specified range.

Description: The secondary voltage is monitored and a power-down is initiated, or there is a switch-over to a second power unit, if it is not in its specified range.

A.8.3 Power-down with safety shut-off

NOTE This technique/measure is referenced in Table A.9 of IEC 61508-2.

Aim: To shut off the power with all safety critical information stored.

Description: Overvoltage or undervoltage is detected early enough so that the internal state can be saved in non-volatile memory (if necessary), and so that all outputs can be set to a safe condition by the power-down routine, or that all outputs can be switched to a safe condition by the power-down routine, or there is a switch-over to a second power unit.

A.9 Temporal and logical program sequence monitoring

NOTE This group of techniques and measures is referenced in Tables A.15, A.16 and A.18 of IEC 61508-2.

Global objective: To detect a defective program sequence. A defective program sequence exists if the individual elements of a program (for example software modules, subprograms or

commands) are processed in the wrong sequence or period of time, or if the clock of the processor is faulty.

A.9.1 Watch-dog with separate time base without time-window

NOTE This technique/measure is referenced in Tables A.10 and A.11 of IEC 61508-2.

Aim: To monitor the behaviour and the plausibility of the program sequence.

Description: External timing elements with a separate time base (for example watch-dog timers) are periodically triggered to monitor the computer's behaviour and the plausibility of the program sequence. It is important that the triggering points are correctly placed in the program. The watch-dog is not triggered at a fixed period, but a maximum interval is specified.

A.9.2 Watch-dog with separate time base and time-window

NOTE This technique/measure is referenced in Tables A.10 and A.11 of IEC 61508-2.

Aim: To monitor the behaviour and the plausibility of the program sequence.

Description: External timing elements with a separate time base (for example watch-dog timers) are periodically triggered to monitor the computer's behaviour and the plausibility of the program sequence. It is important that the triggering points are correctly placed in the program. A lower and upper limit is given for the watch-dog timer. If the program sequence takes a longer or shorter time than expected, emergency action is taken.

A.9.3 Logical monitoring of program sequence

NOTE This technique/measure is referenced in Tables A.10 and A.11 of IEC 61508-2.

Aim: To monitor the correct sequence of the individual program sections.

Description: The correct sequence of the individual program sections is monitored using software (counting procedure, key procedure) or using external monitoring facilities. It is important that the checking points are placed in the program correctly.

A.9.4 Combination of temporal and logical monitoring of program sequences

NOTE This technique/measure is referenced in Tables A.10 and A.11 of IEC 61508-2.

Aim: To monitor the behaviour and the correct sequence of the individual program sections.

Description: A temporal facility (for example a watch-dog timer) monitoring the program sequence is retriggered only if the sequence of the program sections is also executed correctly.

A.9.5 Temporal monitoring with on-line check

NOTE This technique/measure is referenced in Tables A.10 and A.11 of IEC 61508-2.

Aim: To detect faults in the temporal monitoring.

Description: The temporal monitoring is checked at start-up, and a start is only possible if the temporal monitoring operates correctly. For example, a heat sensor could be checked by a heated resistor at start-up.

A.10 Ventilation and heating

NOTE This group of techniques and measures is referenced in Tables A.16 and A.18 of IEC 61508-2.

Global objective: To control failures in the ventilation or heating, and/or their monitoring, if this is safety-related.

A.10.1 Temperature sensor

Aim: To detect over- or under-temperature before the system begins to operate outside specification.

Description: A temperature sensor monitors temperature at the most critical points of the E/E/PE safety-related system. Before the temperature leaves the specified range, emergency action is taken.

A.10.2 Fan control

Aim: To detect incorrect operation of the fans.

Description: The fans are monitored for correct operation. If a fan is not working properly, maintenance (or ultimately, emergency) action is taken.

A.10.3 Actuation of the safety shut-off via thermal fuse

Aim: To shut off the safety-related system before the system works outside of its thermal specification.

Description: A thermal fuse is used to shut off the safety-related system. For a PES, the shut-off is introduced by a power-down routine which stores all information necessary for emergency action.

A.10.4 Staggered message from thermo-sensors and conditional alarm

Aim: To indicate that the safety-related system is working outside its thermal specification.

Description: The temperature is monitored and an alarm is raised if the temperature is outside of a specified range.

A.10.5 Connection of forced-air cooling and status indication

Aim: To prevent overheating by forced-air cooling.

Description: The temperature is monitored and forced-air cooling is introduced if the temperature is higher than a specified limit. The user is informed of the status.

A.11 Communication and mass-storage

Global objective: To control failures during communication with external sources and mass-storage.

A.11.1 Separation of electrical energy lines from information lines

NOTE This technique/measure is referenced in Table A.16 of IEC 61508-2.

Aim: To minimise cross-talk induced by high currents in the information lines.

Description: Electrical energy supply lines are separated from the lines carrying the information. The electrical field which could induce voltage spikes on the information lines decreases with distance.

A.11.2 Spatial separation of multiple lines

NOTE This technique/measure is referenced in Tables A.16 of IEC 61508-2.

Aim: To minimise cross-talk induced by high currents in multiple lines.

Description: Lines carrying duplicated signals are separated from each other. The electrical field which could induce voltage spikes on the multiple lines decreases with the distance. This measure also reduces common cause failures.

A.11.3 Increase of interference immunity

NOTE This technique/measure is referenced in Tables A.16 and A.18 of IEC 61508-2.

Aim: To minimise electromagnetic interference on the safety-related system.

Description: Design techniques such as shielding and filtering are used to increase the interference immunity of the safety-related system to electromagnetic disturbances which may be radiated or conducted on power or signal lines, or result from electrostatic discharges.

NOTE See [16] and [17] for immunity requirements for safety-related systems and for equipment intended to perform safety-related functions (functional safety) in industrial applications.

References:

IEC/TR 61000-5-2:1997, *Electromagnetic compatibility (EMC) – Part 5: Installation and mitigation guidelines – Section 2: Earthing and cabling*

Principles and Techniques of Electromagnetic Compatibility, Second Edition, C. Christopoulos, CRC Press, 2007, ISBN-10: 0849370353, ISBN-13: 978-0849370359

Noise Reduction Techniques in Electronic Systems. H. W. Ott, John Wiley Interscience, 2nd Edition, 1988

EMC for Product Designers. T. Williams, Newnes, 2007, ISBN 0750681705

Grounding and Shielding Techniques in Instrumentation, 3rd edition, R. Morrison . Wiley-Interscience, New York, 1986, ISBN-10: 0471838055, ISBN-13: 978-0471838050

A.11.4 Antivalent signal transmission

NOTE This technique/measure is referenced in Tables A.7 and A.16 of IEC 61508-2.

Aim: To detect the same induced voltages in multiple signal transmission lines.

Description: All duplicated information is transmitted with antivalent signals (for example logic 1 and 0). Common cause failures (for example by electromagnetic emission) can be detected by an antivalent comparator.

Reference:

Elektronik in der Sicherheitstechnik. H. Jürs, D. Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch. 20. Lfg. V/93, Erich Schmidt Verlag, Bielefeld. http://www.bgia-handbuchdigital.de/330220

A.12 Sensors

Global objective: To control failures in the sensors of the safety-related system.

A.12.1 Reference sensor

NOTE This technique/measure is referenced in Table A.13 of IEC 61508-2.

Aim: To detect the incorrect operation of a sensor.

Description: An independent reference sensor is used to monitor the operation of a process sensor. All input signals are checked at suitable time intervals by the reference sensor to detect failures of the process sensor.

Reference:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

A.12.2 Positive-activated switch

NOTE This technique/measure is referenced in Table A.13 of IEC 61508-2.

Aim: To open a contact by a direct mechanical connection between switch cam and contact.

Description: A positive-activated switch opens its normally closed contacts by a direct mechanical connection between switch cam and contact. This ensures that whenever the switch cam is in the operated position, the switch contacts must be open.

Reference:

Verriegelung beweglicher Schutzeinrichtungen. F. Kreutzkampf, K. Becker, Sicherheitstechnisches Informations- und Arbeitsblatt 330210, BIA-Handbuch. 1. Lfg. IX/85, Erich Schmidt Verlag, Bielefeld

A.13 Final elements (actuators)

Global objective: To control failures in the final elements in the safety-related system.

A.13.1 Monitoring

NOTE This technique/measure is referenced in Table A.14 of IEC 61508-2.

Aim: To detect the incorrect operation of an actuator.

Description: The operation of the actuator is monitored (for example by the positively activated contacts of a relay, see monitoring of relay contacts in A.1.2). The redundancy introduced by this monitoring can be used to trigger emergency action.

References:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

Zusammenstellung und Bewertung elektromechanischer Sicherheitsschaltungen für Verriegelungseinrichtungen. F. Kreutzkampf, W. Hertel, Sicherheitstechnisches Informations- und Arbeitsblatt 330212, BIA-Handbuch. 17. Lfg. X/91, Erich Schmidt Verlag, Bielefeld

A.13.2 Cross-monitoring of multiple actuators

NOTE This technique/measure is referenced in Table A.14 of IEC 61508-2.

Aim: To detect faults in actuators by comparing the results.

Description: Each multiple actuator is monitored by a different hardware channel. If a discrepancy occurs, emergency action is taken.

A.14 Measures against the physical environment

NOTE This technique/measure is referenced in Tables A.16 and A.18 of IEC 61508-2

Aim: To prevent influences of the physical environment (water, dust, corrosive substances) causing failures.

Description: The enclosure of the equipment is designed to withstand the expected environment.

Reference:

IEC 60529:1989, *Degrees of protection provided by enclosures (IP Code)*

Annex B
(informative)

Overview of techniques and measures for E/E/PE safety related systems: avoidance of systematic failures (see IEC 61508-2 and IEC 61508-3)

NOTE Many techniques in this annex are applicable to software but have not been duplicated in Annex C.

B.1 General measures and techniques

B.1.1 Project management

NOTE This technique/measure is referenced in Tables B.1 to B.6 of IEC 61508-2.

Aim: To avoid failures by adoption of an organisational model and rules and measures for development and testing of safety-related systems.

Description: The most important and best measures are

- the creation of an organisational model, especially for quality assurance which is set down in a quality assurance handbook; and
- the establishment of regulations and measures for the creation and validation of safety-related systems in cross-project and project-specific guidelines.

A number of important basic principles are set down in the following:

- definition of a design organisation:
  - tasks and responsibilities of the organisational units,
  - authority of the quality assurance departments,
  - independence of quality assurance (internal inspection) from development;
- definition of a sequence plan (activity models):
  - determination of all activities which are relevant during execution of the project including internal inspections and their scheduling,
  - project update;
- definition of a standardised sequence for an internal inspection:
  - planning, execution and checking of the inspection (inspection theory),
  - releasing mechanisms for subproducts,
  - the safekeeping of repeat inspections;
- configuration management:
  - administration and checking of versions,
  - detection of the effects of modifications,
  - consistency inspections after modifications;
- introduction of a quantitative assessment of quality assurance measures:
  - requirement acquisition,
  - failure statistics;
- introduction of computer-aided universal methods, tools and training of personnel.

References:

ISO 9001:2008, *Quality management systems – Requirements*

ISO/IEC 15504 (all parts), *Information technology – Process assessment*

CMMI: Guidelines for Process Integration and Product Improvement, 2nd Edition. M. B. Chrissis, M. Konrad, S. Shrum, Addison-Wesley Professional, 2006, ISBN-10: 0-3212-7967-0, ISBN-13: 978-0-3212-7967-5

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

Dependability of Critical Computer Systems 1. F. J. Redmill, Elsevier Applied Science, 1988, ISBN 1-85166-203-0

B.1.2 Documentation

NOTE 1 This technique/measure is referenced in Tables B.1 to B.6 of IEC 61508-2.

NOTE 2 See also Clause 5 and Annex A of IEC 61508-1.

Aim: To avoid failures and facilitate assessment of system safety, by documenting each step during development.

Description: The operational capacity and safety, as well as the care taken in development by all parties involved, has to be demonstrated during assessment. In order to be able to show the development care, and in order to guarantee the verification of the evidence of safety at any time, special importance is given to the documentation.

Important common measures are the introduction of guidelines and computer aid, i.e.

- guidelines, which
  - specify a grouping plan;
  - ask for checklists for the contents; and
  - determine the form of the document;
- administration of the documentation with the help of a computer-aided and organised project library.

Individual measures are:

- separation in the documentation
  - of the requirements,
  - of the system (user-documentation) and
  - of the development (including internal inspection);
- grouping of the development documentation according to the safety lifecycle;
- definition of standardised documentation modules, from which the documents can be compiled;
- clear identification of the constituent parts of the documentation;
- formalised revision update;
- selection of clear and intelligible means of description:
  - formal notation for determinations;
  - natural language for introductions, justifications and representations of intentions;

- graphical representations for examples;
- semantic definition of graphical elements; and
- directories of specialised words.

References:

IEC 61506:1997, *Industrial-process measurement and control – Documentation of application software*

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.1.3 Separation of E/E/PE system safety functions from non-safety functions

NOTE This technique/measure is referenced in Tables B.1 and B.6 of IEC 61508-2.

Aim: To prevent the non-safety-related part of the system from influencing the safety-related part in undesired ways.

Description: In the specification, it should be decided whether a separation of the safety-related systems and non-safety-related systems is possible. Clear specifications should be written for the interfacing of the two parts. A clear separation reduces the effort for testing the safety-related systems.

Reference:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.1.4 Diverse hardware

NOTE This technique/measure is referenced in Tables A.15, A.16 and A.18 of IEC 61508-2.

Aim: To detect systematic failures during operation of the EUC, using diverse components with different rates and types of failures.

Description: Different types of components are used for the diverse channels of a safety-related system. This reduces the probability of common cause failures (for example overvoltage, electromagnetic interference), and increases the probability of detecting such failures.

Existence of different means of performing a required function, for example different physical principles, offer other ways of solving the same problem. There are several types of diversity. Functional diversity employs the use of different approaches to achieve the same result.

Reference :

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.2 E/E/PE system design requirements specification

Global objective: To produce a specification which is, as far as possible, complete, free from mistakes, free from contradiction, and simple to verify.

B.2.1 Structured specification

NOTE This technique/measure is referenced in Tables B.1 and B.6 of IEC 61508-2.

Aim: To reduce complexity by creating a hierarchical structure of partial requirements. To avoid interface failures between the requirements.

Description: This technique structures the functional specification into partial requirements such that the simplest possible, visible relations exist between the latter. This analysis is successively refined until small clear partial requirements can be distinguished. The result of the final refinement is a hierarchical structure of partial requirements which provide a framework for the specification of the complete requirements. This method emphasises the interfaces of the partial requirements and is particularly effective for avoiding interface failures.

References:

ESA PSS 05-02, *Guide to the user requirements definition phase*, Issue 1, Revision 1, ESA Board for Software Standardisation and Control (BSSC), ESA, Paris, March 1995, ftp://ftp.estec.esa.nl/pub/wm/wme/bssc/PSS0502.pdf

Structured Analysis and System Specification. T. De Marco, Yourdon Press, Englewood Cliffs, 1979, ISBN-10: 0138543801, ISBN-13: 978-0138543808

B.2.2 Formal methods

NOTE 1 See C.2.4 for details of specific formal methods.

NOTE 2 This technique/measure is referenced in Tables B.1, B.2 and B.6 of IEC 61508-2.

Aim: Formal methods transfers the principles of mathematical reasoning to the specification and implementation of technical systems therefore increase the completeness, consistency or correctness of a specification or implementation.

Description: Formal methods provide a means of developing a description of a system during specification and/or implementation phase. These formal descriptions are mathematical models of the system function and/or structure.

Therefore unambiguous system description could be achieved (e.g. any state of an automaton is described by its initial state, inputs and the transition equations of the automaton) which increase understanding of the underlying system.

Choosing a suitable formal method is a difficult undertaking requiring full understanding of system, its development process and the range of mathematical models that could possibly be used (see following notes).

NOTE 3 The theorems of interest of the model (properties) represent guaranties about the system which provides far more confidence than simulation i.e. observing selected actions of the system.

NOTE 4 The disadvantages of formal methods can be:

– Fixed level of abstraction;
– limitations to capture all functionality that is relevant at the given stage;
– difficulty that implementation engineers have to understand the model;
– high efforts to develop, analyze and maintain model over the lifecycle of system;
– availability of efficient tools which support the building and analysis of model;
– availability of staff capable to develop and analyze model.

NOTE 5 The formal methods community's focus clearly was been the modeling of the target function of system often deemphasizing the fault robustness of a system. Therefore respective formal methods including system robustness has to be selected.

Reference:

Formal Specification: Techniques and Applications. N.Nissanke, Springer-Verlag Telos, 1999, ISBN-10: 1852330023

B.2.3 Semi-formal methods

NOTE 1 IEC 61508- 3 Table B.7 extends this Annex B list with other semi-formal software related techniques. art 3 lists:

– logic/function block diagrams: described in IEC 61131-3;
– sequence diagrams: described in IEC 61131-3;
– data flow diagrams: see C.2.2;
– finite state machines/state transition diagrams: see B.2.3.2;
– time Petri nets: see B.2.3.3;
– entity-relationship-attribute Data models: see B.2.4.4;
– message sequence charts: see C.2.14;
– decision/truth tables: see C.6.1.

Aim: To express parts of a specification unambiguously and consistently, so that some mistakes, omissions and wrong behaviour can be detected.

NOTE 2 This technique/measure is referenced in Tables B.1, B.2 and B.6 of IEC 61508-2 and in Tables A.1, A.2, A.4, B.7, C.1, C.2, C.4 and C.17 of IEC 61508-3.

B.2.3.1 General

Aim: To prove that the design meets its specification.

Description: Semi-formal methods provide a means of developing a description of a system at some stage in its development, i.e. specification, design or coding. The description can in some cases be analysed by machine or animated to display various aspects of the system behaviour. Animation can give extra confidence that the system meets the real requirement as well as the specified requirement.

Two semi-formal methods are described in the following subclauses.

B.2.3.2 Finite state machines / state transition diagrams

NOTE This technique/measure is referenced in Tables B.5, B.7, C.15 and C.17 of IEC 61508-3.

Aim: To model, verify, specify or implement the control structure of a system.

Description: Many systems can be described 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 describing a system's actions for every input in every state we can describe a system completely. The resulting model of the system is called a finite state machine (or finite state automata). 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 receiving the input when in the given state.

Where a system is complicated or has a natural structure, this can be reflected in a layered finite state machine. A Statechart is a type of state transition diagram in which nested states are allowed (the object state splits into two or more sub-states which can evolve in parallel, and possibly recombine into a single state at some point); this adds to the expressive power of the state transition notation but can add extra complexity which may be undesirable in a safety related system. Statecharts have a formal (mathematical) specification. State transition diagrams can apply to the whole system or to some object or element within it.

A specification or design expressed as a finite state machine can be checked for

- completeness (the system or object must have an action and new state for every input in every state);
- consistency (only one state transition is possible 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); and
- absence of endless loops or dead-end states; etc.

These are important properties for critical systems. Tools to support these checks are easily developed and various models based on finite state automata (formal languages, Petri nets, Markov graphs, etc.) can be used. Algorithms also exist that allow the automatic generation of test cases for verifying a finite state machine implementation or for animating a finite state machine model. State transition diagrams and Statecharts are widely supported by tools which allow the diagrams to be drawn and checked, and which will generate code to implement the described state machine.

They can also be used for failure probability calculations, see B.6 and C.6.

References:

Introduction to Automata Theory, Languages, and Computation (3rd Edition). J. Hopcroft, R. Motwani, J. Ullman, Addison-Wesley Longman Publishing Co, 2006, ISBN: 0321462254

Sécurisation des architectures informatiques. Jean-Louis Boulanger, Hermès – Lavoisier, 2009, ISBN: 978-2-7462-1991-5

B.2.3.3 Time Petri nets

NOTE This technique/measure is referenced in Tables B.5, B.7, C.15 and C.17 of IEC 61508-3.

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 are a particular case of finite state automata. They belong to a class of graph theoretic models which are suitable for representing information and control flow in systems that exhibit concurrency and have 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 places to the transition become unmarked, and each output place from the transition is marked instead.

The potential hazards can be represented as particular states (markings) in the model. The Petri net model can be extended to allow for timing features of the system. Although "classical" Petri nets concentrate on control flow aspects, several extensions have been proposed to incorporate data flow into the model.

They also provide a very efficient support for performing Monte Carlo simulation in order to achieve failure probability calculations, see B.6.6.8.

References:

Timed Petri Nets: Theory and Application. Jiacun Wang, Springer, 1998, ISBN 0792382706

Sécurisation des architectures informatiques. Jean-Louis Boulanger, Hermès – Lavoisier, 2009, ISBN: 978-2-7462-1991-5

B.2.4 Computer-aided specification tools

NOTE This technique/measure is referenced in Tables B.1 and B.6 of IEC 61508-2 and in Tables A.1, A.2, C.1 and C.2 of IEC 61508-3.

B.2.4.1 General

Aim: To use formal specification techniques to facilitate automatic detection of ambiguity and completeness.

Description: The technique produces a specification in the form of a database that can be automatically inspected to assess consistency and completeness. The specification tool can animate various aspects of the specified system to the user. In general, the technique supports not only the creation of the specification but also of the design and other phases of the project lifecycle. Specification tools can be classified according to the following subclauses.

B.2.4.2 Tools oriented towards no specific method

Aim: To help the user write a good specification by providing prompts and links between related parts.

Description: The specification tool takes over some routine work from the user and supports the project management. It does not enforce any particular specification methodology. The relative independence with regard to method allows users a great deal of freedom but also gives them little of the specialised support necessary when creating specifications. This makes familiarisation with the system more difficult.

B.2.4.3 Model orientated procedure with hierarchical analysis

Aim: To avoid incompleteness, ambiguity and contradiction in the specification, e.g. supporting the user writing a good specification by ensuring consistency between descriptions of actions and data at various levels of abstraction.

Description: This method gives a functional representation of the desired system (structured analysis) at various levels of abstraction (degree of precision). There is a huge arsenal of such models: finite automata are a class of such models widely used to describe the evolution of discrete/digital systems. Differential equations are similar in spirit and aim at continuous/analogue systems. The analysis at various levels acts on both actions and data. Assessment of ambiguity and completeness is possible between hierarchical levels as well as between two functional units (modules) on the same level (e.g. any state of a system model is described by its initial state, inputs and the transition equations of the automaton).

NOTE Issues of model based descriptions may be the level of abstraction, limitations to capture all functionality that is relevant at the given stage, difficulty that practitioners have to understand the model (from reading the syntax to understanding), high efforts to develop, analyze and maintain a model over the lifecycle of a system, availability of efficient tools which support the building and analysis of model (development of such tools is certainly a high effort undertaking) and availability of staff capable to develop and analyze models.

Reference:

System requirements analysis. Jeffrey O. Grady, Academic Press, 2006, ISBN 012088514X, 9780120885145

B.2.4.4 Entity-relationship-attribute data models

Aim: To help the user write a good specification by focusing on entities within the system and relationships between them.

Description: The desired system is described as a collection of objects and their relationships. The tool enables one to determine which relationships can be interpreted by the

system. In general, the relationships permit a description of the hierarchical structure of the objects, the data flow, the relationships between the data, and which data are subject to certain manufacturing processes. The classical procedure has been extended for process control applications. Inspection capabilities and support for the user depend on the variety of relationships illustrated. On the other hand, a large number of representation possibilities makes the application of this technique complex.

Reference:

Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle. Karl Eugene Wiegers, Microsoft Press, 2003, ISBN 0735618798, 9780735618794

B.2.4.5 Incentive and answer

Aim: To help the user write a good specification by identifying stimulus-response relationships.

Description: The relationships between the objects of the system are specified in a notation of "incentives" and "answers". A simple and easily expanded language is used which contains language elements which represent objects, relationships, characteristics and structures.

B.2.5 Checklists

NOTE This technique/measure is referenced in Tables B.1, B.2 and B.6 of IEC 61508-2 and in Tables A.10, B.8, C.10 and C.18 of IEC 61508-3.

Aim: To draw attention to, and manage critical appraisal of, all important aspects of the system by safety lifecycle phase, ensuring comprehensive coverage without laying down exact requirements.

Description: A set of questions to be answered 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. Checklists can be used for all phases of the overall, E/E/PE system safety and software safety lifecycles and are particularly useful as a tool to aid the functional safety assessment.

To accommodate wide variations in 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 responses should be used to document the results for each question. This conciseness greatly simplifies the procedure of reaching an overall conclusion as to the results of the checklist assessment.

References:

IEC 60880:2006, *Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions*

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

Software Quality Assurance: From Theory to Implementation. Daniel Galin, Pearson Education, 2004, ISBN 0201709457, 9780201709452

IEC 61346 (all parts), *Industrial systems, installations and equipment and industrial products – Structuring principles and reference designation*

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

Risk Assessment and Risk Management for the Chemical Process Industry. H.R. Greenberg, J.J. Cramer, John Wiley and Sons, 1991, ISBN 0471288829, 9780471288824

B.2.6 Inspection of the specification

NOTE This technique/measure is referenced in Tables B.1 and B.6 of IEC 61508-2.

Aim: To avoid incompleteness and contradiction in the specification.

Description: Inspection is a general technique in which various qualities of a specification document are assessed by an independent team. The inspection team puts questions to the creator, who must answer them satisfactorily. The examination should (if possible) be carried out by a team that was not involved in the creation of the specification. The required degree of independence is determined by the safety integrity levels demanded of the system. The independent inspectors should be able to reconstruct the operational function of the system in an indisputable manner without referring to any further specifications. They must also check that all relevant safety and technical aspects in the operational and organisational measures are covered. This procedure has proved itself to be very effective in practice.

References:

IEC 61160:2005, *Design review*

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

Software Quality Assurance: From Theory to Implementation. D. Galin, Pearson Education, 2004, ISBN 0201709457, 9780201709452

B.3 E/E/PE system design and development

Global objective: To produce a stable design of the safety-related system in conformance with the specification.

B.3.1 Observance of guidelines and standards

NOTE This technique/measure is referenced in Table B.2 of IEC 61508-2.

Aim: To observe application sector standards (not specified in this standard).

Description: Guidelines should be complied with during the design of the safety-related system. These guidelines should firstly lead to safety-related systems which are practically

free from failures, and secondly facilitate the subsequent safety validation. They can be universally valid, specific to a project, or specific only to a single phase.

References:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.3.2 Structured design

NOTE This technique/measure is referenced in Tables B.2 and B.6 of IEC 61508-2.

Aim: To reduce complexity by creating a hierarchical structure of partial requirements. To avoid interface failures between the requirements. To simplify verification.

Description: When designing the hardware, specific criteria or methods should be used. For example, the following might be required:

– a hierarchically structured circuit design;

– use of manufactured and tested circuit parts.

Similarly, when designing the software, the use of structure charts enables an unambiguous structure of the software modules to be created. This structure shows how the modules relate to each other, the precise data which passes between modules, and the precise controls that exist between modules.

References:

IEC 61346 (all parts), *Industrial systems, installations and equipment and industrial products – Structuring principles and reference designation*

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

Software Design. D. Budgen, Pearson Education, 2003, ISBN 0201722194, 9780201722192

An Overview of JSD, J. R. Cameron, IEEE Trans SE-12 No. 2, February 1986

Structured Development for Real-Time Systems (3 Volumes). P. T. Yourdon, P. T. Yourdon Press, 1985

Structured Development for Real-Time Systems (3 Volumes). P. T. Ward, S. J. Mellor, Yourdon Press, 1985

Applications and Extensions of SADT. D. T. Ross, Computer, 25-34, April 1985

Essential Systems Analysis. St. M. McMenamin, F. Palmer, Yourdon Inc, 1984

Structured Analysis (SA): A language for communicating ideas. D. T. Ross, IEEE Trans. Software Eng, Vol. SE-3 (1), 16-34

B.3.3 Use of well-tried components

NOTE This technique/measure is referenced in Tables B.2 and B.6 of IEC 61508-2.

Aim: To reduce the risk of numerous first time and undetected faults by the use of components with specific characteristics.

Description: The selection of well-tried components is carried out by the manufacturer, with regard to safety according to the reliability of the elements (for example the use of operationally tested physical units to meet high safety requirements, or the storing of safety-

related programs in safe memories only). The safety of memories can refer to unauthorised access as well as environmental influences (electromagnetic compatibility, radiation, etc) and the response of the elements in the event of a failure occurring.

References:

IEC 61163-1:2006, *Reliability stress screening – Part 1: Repairable assemblies manufactured in lots*

B.3.4 Modularisation

NOTE This technique/measure is referenced in Tables B.2 and B.6 of IEC 61508-2.

Aim: To reduce complexity and avoid failures, related to interfacing between subsystems.

Description: Every subsystem, at all levels of the design, is clearly defined and is of restricted size (only a few functions). The interfaces between subsystems are kept as simple as possible and the cross-section (i.e. shared data, exchange of information) is minimised. The complexity of individual subsystems is also restricted.

References:

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

Software Reliability – Principles and Practices. G. J. Myers, Wiley-Interscience, New York, 1976, ISBN-10: 0471627658, ISBN-13: 978-0471627654

B.3.5 Computer-aided design tools

NOTE This technique/measure is referenced in Tables B.2 and B.6 of IEC 61508-2 and in Tables A.4 and C.4 of IEC 61508-3.

Aim: To carry out the design procedure more systematically. To include appropriate automatic construction elements which are already available and tested.

Description: Computer-aided design tools (CAD) should be used during the design of both hardware and software when available and justified by the complexity of the system. The correctness of such tools should be demonstrated by specific testing, by an extensive history of satisfactory use, or by independent verification of their output for the particular safety-related system that is being designed.

Support tools should be selected for their degree of integration. In this context, tools are integrated if they work co-operatively such that the outputs from one tool have suitable content and format for automatic input to a subsequent tool, thus minimizing the possibility of introducing human error in the reworking of intermediate results.

References:

Overview of Technology Computer-Aided Design Tools and Applications in Technology Development, *Manufacturing and Design*. W. Fichtner, Journal of Computational and Theoretical Nanoscience, Volume 5, Number 6, June 2008, pp. 1089-1105(17)

The Electromagnetic Data Exchange: Much more than a Common Data Format. P.E. Frandsen et al. In *Proceeding of the 2nd European Conference on Antennas and Propagation*. The Institution of Engineering and Technology (IET), 2007, ISBN 978-0-86341-842-6

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

B.3.6 Simulation

NOTE This technique/measure is referenced in Tables B.2, B.5 and B.6 of IEC 61508-2.

Aim: To carry out a systematic and complete inspection of an electrical/electronic circuit, of both the functional performance and the correct dimensioning of the components.

Description: The function of the safety-related system circuit is simulated on a computer via a software behavioural model. Individual components of the circuit each have their own simulated behaviour, and the response of the circuit in which they are connected is examined by looking at the marginal data of each component.

B.3.7 Inspection (reviews and analysis)

NOTE This technique/measure is referenced in Tables B.2 and B.6 of IEC 61508-2.

Aim: To reveal discrepancies between the specification and implementation.

Description: Specified functions of the safety-related system are examined and evaluated to ensure that the safety-related system conforms to the requirements given in the specification. Any points of doubt concerning the implementation and use of the product are documented so they may be resolved. In contrast to a walk-through, the author is passive and the inspector is active during the inspection procedure.

References:

IEC 61160:2005, *Design Review*

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

ANSI/IEE 1028:1997, *IEEE Standard for software reviews*

Dependability of Critical Computer Systems 3. P. G. Bishop et al., Elsevier Applied Science, 1990, ISBN 1-85166-544-7

B.3.8 Walk-through

NOTE This technique/measure is referenced in Tables B.2 and B.6 of IEC 61508-2.

Aim: To reveal discrepancies between the specification and implementation.

Description: Specified functions of the safety-related system draft are examined and evaluated to ensure that the safety-related system complies with the requirements given in the specification. Doubts and potential weak points concerning the realisation and use of the product are documented so that they may be resolved. In contrast to an inspection, the author is active and the inspector is passive during the walk-through.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

ANSI/IEEE 1028:1997, *IEEE Standard for software reviews*

Dependability of Critical Computer Systems 3. P. G. Bishop et al., Elsevier Applied Science, 1990, ISBN 1-85166-544-7

Methodisches Testen von Programmen. G. J. Myers, Oldenbourg Verlag, München, Wien, 1987

B.4 E/E/PE system operation and maintenance procedures

Global objective: To develop procedures which help to avoid failures during the operation and maintenance of the safety-related system.

B.4.1 Operation and maintenance instructions

NOTE This technique/measure is referenced in Table B.4 of IEC 61508-2.

Aim: To avoid mistakes during operation and maintenance of the safety-related system.

Description: User instructions contain essential information on how to use and how to maintain the safety-related system. In special cases, these instructions will also include examples on how to install the safety-related system in general. All instructions must be easily understood. Figures and schematics should be used to describe complex procedures and dependencies.

Reference: *Guidelines for Safe Automation of Chemical Processes*. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.4.2 User friendliness

NOTE This technique/measure is referenced in Table B.4 of IEC 61508-2.

Aim: To reduce complexity during operation of the safety-related system.

Description: The correct operation of the safety-related system may depend to some degree on human operation. By considering the relevant system design and the design of the workplace, the safety-related system developer must ensure that

- the need for human intervention is restricted to an absolute minimum;
- the necessary intervention is as simple as possible;
- the potential for harm from operator error is minimised;
- the intervention facilities and indication facilities are designed according to ergonomic requirements;
- the operator facilities are simple, well labelled and intuitive to use;
- the operator is not overstrained, even in extreme situations;
- training on intervention procedures and facilities is adapted to the level of knowledge and motivation of the trainee user.

B.4.3 Maintenance friendliness

NOTE This technique/measure is referenced in Table B.4 of IEC 61508-2.

Aim: To simplify maintenance procedures of the safety-related system and to design the necessary means for effective diagnosis and repair.

Description: Preventive maintenance and repair is often carried out under difficult circumstances and under pressure from deadlines. Therefore, the safety-related system developer should ensure that

- safety-related maintenance measures are necessary as seldom as possible or even, ideally, not necessary at all;
- sufficient, sensible and easy-to-handle diagnosis tools are included for those repairs that are unavoidable – tools should include all necessary interfaces;
- if separate diagnosis tools have to be developed or obtained, then these should be available on time.

B.4.4 Limited operation possibilities

NOTE This technique/measure is referenced in Tables B.4 and B.6 of IEC 61508-2.

Aim: To reduce the operation possibilities for the normal user.

Description: This approach reduces the operation possibilities by

- limiting the operation within special operating modes, for example by key switches;
- limiting the number of operating elements;
- limiting the number of generally possible operating modes.

Reference:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.4.5 Operation only by skilled operators

NOTE This technique/measure is referenced in Tables B.4 and B.6 of IEC 61508-2.

Aim: To avoid operating failures caused by misuse.

Description: The safety-related system operator is trained to a level which is appropriate to the complexity and safety integrity level of the safety-related system. Training includes studying the background of the production process and knowing the relationship between the safety-related system and the EUC.

Reference:

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.4.6 Protection against operator mistakes

NOTE This technique/measure is referenced in Tables B.4 and B.6 of IEC 61508-2.

Aim: To protect the system against all classes of operator mistakes.

Description: Wrong inputs (value, time, etc) are detected via plausibility checks or monitoring of the EUC. To integrate these facilities into the design, it is necessary to state at a very early stage which inputs are possible and which are permissible.

B.4.7 (Not used)

B.4.8 Modification protection

NOTE This technique/measure is referenced in Tables A.17 and A.18 of IEC 61508-2.

Aim: To protect the safety-related system against hardware modifications by technical means.

Description: Modifications or manipulations are detected automatically, for example by plausibility checks for the sensor signals, detection by the technical process and by automatic start-up tests. If a modification is detected, then emergency action is taken.

B.4.9 Input acknowledgement

NOTE This technique/measure is referenced in Tables A.17 and A.18 of IEC 61508-2.

Aim: A mistake during operation is detected by the operator himself before activating the EUC.

Description: An input to the EUC via the safety-related system is echoed to the operator before being sent to the EUC so that the operator has the possibility to detect and correct a mistake. As well as abnormal, unprovoked personnel action, the system design should consider top/bottom speed limits and direction of human reaction. This would avoid, for example, the operator pressing keys faster than expected, causing the system to read a double keystroke as a single one, or a key to be pressed twice because the system (display) was too slow to react to the first instance. The same key stroke should not be valid more than once in succession for critical data entry; pressing the "enter" or "yes" key unlimited times must not lead to an unsafe action of the system.

Time-out procedures should be included with multiple choice questions (yes/no, etc.) to cater for when the operator may not make up his mind and leave the system waiting.

Ability to reboot a safety-related PES makes the system vulnerable unless both software/hardware are designed with such occasions in mind.

B.5 E/E/PE system integration

Global objective: To avoid failures during the integration phase and to reveal any failures that are made during this and previous phases.

B.5.1 Functional testing

NOTE This technique/measure is referenced in Tables B.3 and B.5 of IEC 61508-2 and in Tables A.5, A.6, A.7, C.5, C.6 and C.7 of IEC 61508-3.

Aim: To reveal failures during the specification and design phases. To avoid failures during implementation and the integration of software and hardware.

Description: During the functional tests, reviews are carried out to see whether the specified characteristics of the system have been achieved. The system is given input data which adequately characterises the normally expected operation. The outputs are observed and their response is compared with that given by the specification. Deviations from the specification and indications of an incomplete specification are documented.

Functional testing of electronic components designed for a multi-channel architecture usually involves the manufactured components being tested with pre-validated partner components. In addition to this, it is recommended that the manufactured components be tested in combination with other partner components of the same batch, in order to reveal common mode faults which would otherwise have remained masked.

Also, the working capacity of the system has to be sufficient, see guidance in C.5.20.

References:

Software Testing and Quality Assurance. K. Naik, P. Tripathy, Wiley Interscience, 2008, Print ISBN: 9780471789116 Online ISBN: 9780470382844

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

Practical Software Testing: A Process-oriented Approach. I. Burnstein, Springer, 2003, ISBN 0387951318, 9780387951317

Dependability of Critical Computer Systems 3. P. G. Bishop et al., Elsevier Applied Science, 1990, ISBN 1-85166-544-7

B.5.2 Black-box testing

NOTE This technique/measure is referenced in Tables B.3, B.5 and B.6 of IEC 61508-2 and in Tables A.5, A.6, A.7, C.5, C.6 and C.7 of IEC 61508-3.

Aim: To check the dynamic behaviour under real functional conditions. To reveal failures to meet functional specification, and to assess utility and robustness.

Description: The functions of a system or program are executed in a specified environment with specified test data which is derived systematically from the specification according to established criteria. This exposes the behaviour of the system and permits a comparison with the specification. No knowledge of the internal structure of the system is used to guide the testing. The aim is to determine whether the functional unit carries out correctly all the functions required by the specification. The technique of forming equivalence classes is an example of the criteria for blackbox test data. The input data space is subdivided into specific input value ranges (equivalence classes) with the aid of the specification. Test cases are then formed from the

- data from permissible ranges;
- data from inadmissible ranges;
- data from the range limits;
- extreme values;
- and combinations of the above classes.

Other criteria can be effective in order to select test cases in the various test activities (module test, integration test and system test). For example, the criterion "extreme operational conditions" is relied upon for the system test within the framework of a validation.

References:

Software Testing and Quality Assurance. K. Naik, P. Tripathy, Wiley Interscience, 2008, Print ISBN: 9780471789116 Online ISBN: 9780470382844

Essentials of Software Engineering. Frank F. Tsui, Orlando Karam. Jones & Bartlett, 2006. ISBN 076373537X, 9780763735371

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

Systematic Software Testing. Rick D. Craig, Stefan P. Jaskiel. Artech House, 2002. ISBN 1580535089, 9781580535083

B.5.3 Statistical testing

NOTE This technique/measure is referenced in Tables B.3, B.5 and B.6 of IEC 61508-2.

Aim: To check the dynamic behaviour of the safety-related system and to assess its utility and robustness.

Description: This approach tests a system or program with input data selected according to the expected statistical distribution of the real operating inputs – the operational profile.

References:

A discussion of statistical testing on a safety-related application. S Kuball, J H R May, Proc. IMechE Vol. 221 Part O: J. Risk and Reliability, Institution of Mechanical Engineers, 2007

Practical Reliability Engineering. P. O'Connor, D. Newton, R. Bromley, John Wiley and Sons, 2002, ISBN 0470844639, 9780470844632

Dependability of Critical Computer Systems 3. P. G. Bishop et al., Elsevier Applied Science, 1990, ISBN 1-85166-544-7

Dependability of Critical Computer Systems 1. F. J. Redmill, Elsevier Applied Science, 1988, ISBN 1-85166-203-0

B.5.4 Field experience

NOTE 1 This technique/measure is referenced in Tables B.3, B.5 and B.6 of IEC 61508-2.

NOTE 2 See also C.2.10 for a similar measure and Annex D for a statistical approach, both in the context of software.

Aim: To use field experience from different applications as one of the measures to avoid faults either during E/E/PE system integration and/or during E/E/PE system safety validation.

Description: Use of components or subsystems, which have been shown by experience to have no, or only unimportant, faults when used, essentially unchanged, over a sufficient period of time in numerous different applications. Particularly for complex components with a multitude of possible functions (for example operating system, integrated circuits), the developer shall pay attention to which functions have actually been tested by the field experience. For example, consider self-test routines for fault detection: if no break-down of the hardware occurs within the operating period, the routines cannot be said to have been tested, since they have never performed their fault detection function.

For field experience to apply, the following requirements must have been fulfilled:

- unchanged specification;
- 10 systems in different applications;
- 10^5 operating hours and at least one year of service history.

NOTE 3 Sector standards may specify different numbers.

The field experience is demonstrated through documentation of the vendor and/or operating company. This documentation must contain at least

- the exact designation of the system and its component, including version control for hardware;
- the users and time of application;
- the operating hours;
- the procedures for the selection of the systems and applications procured to the proof;

– the procedures for fault detection and fault registration as well as fault removal.

References:

IEC 60300-3-2:2004, *Dependability management – Part 3-2: Application guide – Collection of dependability data from the field*

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993, ISBN-10: 0-8169-0554-1, ISBN-13: 978-0-8169-0554-6

B.6 E/E/PE system safety validation

Global objective: To prove that the E/E/PE safety-related system conforms to the E/E/PE system safety requirements specification and E/E/PE system design requirements specification.

B.6.1 Functional testing under environmental conditions

NOTE This technique/measure is referenced in Table B.5 of IEC 61508-2.

Aim: To assess whether the safety-related system is protected against typical environmental influences.

Description: The system is put under various environmental conditions (for example according to the standards in the IEC 60068 series or the IEC 61000 series), during which the safety functions are assessed for their reliability (and compatibility with the standards mentioned).

References:

IEC 60068-1:1988, *Environmental testing – Part 1: General and guidance*
Amendment 1(1992)

IEC 61000-4-1:2006, *Electromagnetic compatibility (EMC) – Part 4-1: Testing and measurement techniques – Overview of IEC 61000-4 series*

Dependability of Critical Computer Systems 3. P. G. Bishop et al., Elsevier Applied Science, 1990, ISBN 1-85166-544-7

B.6.2 Interference surge immunity testing

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2.

Aim: To check the capacity of the safety-related system to handle peak surges.

Description: The system is loaded with a typical application program, and all the peripheral lines (all digital, analogue and serial interfaces as well as the bus connections and power supply, etc.) are subjected to standard noise signals. In order to obtain a quantitative statement, it is sensible to approach the surge limit carefully. The chosen class of noise is not complied with if the function fails.

References:

IEC 61000-4-5:2005, *Electromagnetic compatibility (EMC) – Part 4-5: Testing and measurement techniques – Surge immunity test*

C37.90.1-2002, *IEEE Standard for Surge Withstand Capability (SWC) Tests for Relays and Relay Systems Associated with Electric Power Apparatus*

B.6.3 (Not used)

B.6.4 Static analysis

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2 and in Tables A.9, B.8, C.9 and C.18 of IEC 61508-3.

Aim: To avoid systematic faults that can lead to breakdowns in the system under test, either early or after many years of operation.

Description: This systematic and possibly computer-aided approach inspects specific static characteristics of the prototype system to ensure completeness, consistency, lack of ambiguity regarding the requirement in question (for example construction guidelines, system specifications, and an appliance data sheet). A static analysis is reproducible. It is applied to a prototype which has reached a well-defined stage of completion. Some examples of static analysis, for hardware and software, are

- consistency analysis of the data flow (such as testing if a data object is interpreted everywhere as the same value);
- control flow analysis (such as path determination, determination of non-accessible code);
- interface analysis (such as investigation of variable transfer between various software modules);
- dataflow analysis to detect suspicious sequences of creating, referencing and deleting variables;
- testing adherence to specific guidelines (for example creepage distances and clearances, assembly distance, physical unit arrangement, mechanically sensitive physical units, exclusive use of the physical units which were introduced).

References:

Static Analysis and Software Assurance. D. Wagner, Lecture Notes in Computer Science, Volume 2126/2001, Springer, 2001, ISBN 978-3-540-42314-0

An Industrial Perspective on Static Analysis. B A Wichmann, A A Canning, D L Clutterbuck, L A Winsborrow, N J Ward and D W R Marsh. Software Engineering Journal., 69 – 75, March 1995

Dependability of Critical Computer Systems 3. P. G. Bishop et al,. Elsevier Applied Science, 1990, ISBN 1-85166-544-7

B.6.5 Dynamic analysis and testing

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2 and in Tables A.5, A.9, B.2, C.5, C.9 and C.12 of IEC 61508-3.

Aim: To detect specification failures by inspecting the dynamic behaviour of a prototype at an advanced state of completion.

Description: The dynamic analysis of a safety-related system is carried out by subjecting a near-operational prototype of the safety-related system to input data which is typical of the intended operating environment. The analysis is satisfactory if the observed behaviour of the safety-related system conforms to the required behaviour. Any failure of the safety-related system must be corrected and the new operational version must then be reanalysed.

References:

The Concept of Dynamic Analysis. T. Ball, ESEC/FSE ’99, Lecture Notes in Computer Science, Springer, 1999, ISBN 978-3-540-66538-0

Dependability of Critical Computer Systems 3. P. G. Bishop et al,. Elsevier Applied Science, 1990, ISBN 1-85166-544-7

B.6.6 Failure analysis

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2.

B.6.6.1 Failure modes and effects analysis (FMEA)

Aim: To analyse a system design, by examining systematically all possible sources of failure of a system's components and determining the effects of these failures on the behaviour and safety of the system.

Description: The analysis usually takes place through a meeting of engineers. Each component of a system is analysed in turn to give a set of failure modes for the component, their causes and effects (locally and at overall system level), detection procedures and recommendations. If the recommendations are acted upon, they are documented as remedial action taken.

References:

IEC 60812:2006, *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*

Risk Assessment and Risk Management for the Chemical Process Industry. H.R. Greenberg, J.J. Cramer, John Wiley and Sons, 1991, ISBN 0471288829, 9780471288824

Reliability Technology. A. E. Green, A. J. Bourne, Wiley-Interscience, 1972, ISBN 0471324809

B.6.6.2 Cause consequence diagrams

NOTE This technique/measure is referenced in Tables B.3, B.4, C.13 and C.14 of IEC 61508-3.

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

Description: The technique can be regarded as a combination of fault tree and event tree analysis. It starts from a critical (initiating) event and the consequence graph is traced forwards by using YES/NO gates describing success and failure of some operations. This allows building event sequences leading either to an accident or to a mastered situation. Then cause graphs (i.e. fault trees) are built for each failure. Then starting from an accidental situation and going in the backward direction gives a global fault tree with this accidental situation as top event. In the forward direction the possible consequences arising from an event are determined. 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 is defined for use in cause consequence diagrams. The diagrams can be used for generating fault trees and to compute the probability of occurrence of certain critical consequences. It can also be used to produce event trees.

References:

IEC 62502, *Analysis techniques for dependability – Event tree analysis (ETA)*[1]

The Cause Consequence Diagram Method as a Basis for Quantitative Accident Analysis. B. S. Nielsen, Danish Atomic Energy Commission, Riso-M-1374, 1971

---

1 Under consideration.

B.6.6.3 Event tree analysis (ETA)

NOTE This technique/measure is referenced in Tables B.4 and C.14 of IEC 61508-3.

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. An event tree is difficult to build from scratch and using consequence diagram is helpful.

Description: On the top of the diagram is written the sequence conditions that are relevant in the progression of events that follow the initiating event. Starting under the initiating event, which is the target of the analysis, a line is drawn to the first condition in the sequence. There the diagram branches off into "yes" and "no" branches, describing how future events 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. Provided the conditions in the sequences are independent, 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. As conditions are rarely fully independent, such calculation shall be considered cautiously and performed by skilled analysts.

References:

IEC 62502, *Analysis techniques for dependability – Event tree analysis (ETA)*[2]

Risk Assessment and Risk Management for the Chemical Process Industry. H.R. Greenberg, J.J. Cramer, John Wiley and Sons, 1991, ISBN 0471288829, 9780471288824

B.6.6.4 Failure modes, effects and criticality analysis (FMECA)

NOTE Failure analysis is referenced in Tables A.10, B.4, C.10 and C.14 of IEC 61508-3.

Aim: To rank the criticality of components which could result in injury, damage or system degradation through single-point failures, in order to determine which components might need special attention and necessary control measures during design or operation.

Description: This method is comparable to FMEA, but there are one or more columns for indicating the criticality, which can be ranked in many ways. The most laborious method is described by the Society for Automotive Engineers (SAE) in ARP 926. In this procedure, the criticality number for any component is indicated by the number of failures of a specific type expected during each million operations occurring in a critical mode. The criticality number is a function of nine parameters, most of which have to be measured. A very simple method for criticality determination is to multiply the probability of component failure by the damage that could be generated; this method is similar to simple risk factor assessment.

References:

IEC 60812:2006, *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*

Software criticality analysis of COTS/SOUP. P.Bishop, T.Clement, S.Guerra. In Reliability Engineering & System Safety, Volume 81, Issue 3, September 2003, Elsevier Ltd., 2003

Software FMEA techniques. P.L.Goddard. In Proc Annual 2000 Reliability and Maintainability Symposium, IEEE, 2000, ISBN: 0-7803-5848-1

---

2 Under consideration.

B.6.6.5 Fault tree analysis (FTA)

NOTE This technique/measure is referenced in Tables B.4 and C.14 of IEC 61508-3.

Aim: To aid in the analysis of events, or combinations of events, that will lead to a hazard or serious consequence and to perform the probability calculation of the top event.

Description: Starting at an event which would be the immediate cause of a hazard or serious consequence (the "top event"), analysis is carried out in order to identify the causes of this event. This is done in several steps through the use of 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. At the end of the analysis, the fault tree represents the logical function linking the basic events (generally components failures) to the top event (the overall system failure).The technique is mainly intended for the analysis of hardware systems, but there have also been attempts to apply this approach to software failure analysis. This technique can be used qualitatively for failure analysis (identification failure scenarios: minimal cut sets or prime implicants), semi-quantitatively (by ranking scenarios according to their probabilities) and quantitatively for probabilistic calculations of the top event (see C.6).

References:

IEC 61025:2006, *Fault tree analysis (FTA)*

From safety analysis to software requirements. K.M. Hansen, A.P. Ravn, A.P, V Stavridou. IEEE Trans Software Engineering, Volume 24, Issue 7, Jul 1998

B.6.6.6 Markov models

NOTE See B.1 of IEC 61508-6 for the use of this technique against reliability block diagrams, in the context of analysing hardware safety integrity.

Aim: To model the behaviour of the system by a state transition graph and to evaluate probabilistic system parameters (e.g, un-reliability, un-availability, *MTTF*, *MUT*, *MDT*, etc.) of a system.

Description: It is a finite state automaton (see B.2.3.2) represented by a directed graph. The nodes (circles) represent the states and the edges (arrows) between nodes represent the transitions (failure, repairs, etc.) occurring between the states. Edges are weighted with the corresponding failure rates or repair rates. The fundamental property of homogeneous Markov processes is that the future depends only of the present: a change of state, *N*, to a subsequent state, *N*+1, is independent of the previous state, *N*-1. This implies that all the probabilistic laws of the models are exponential.

The failure events, states and rates can be detailed in such a way that a precise description of the system is obtained, for example detected or undetected failures, manifestation of a larger failure, etc. Proof test intervals may also be modelled properly by using the so-called multi-phase Markov processes where the probabilities of the states at the end of one phase (e.g. just before a proof test) can be used to calculate the initial conditions for the next phase (e.g. the probabilities of the various states after a proof test has been performed).

The Markov technique is suitable for modelling multiple 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 lifecycle 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 and some methods of simplification (i.e. reducing the number of states) also exist to handle more complex cases.

Nevertheless, mathematically speaking, a homogeneous Markov graph is only a simple and common set of linear differential equations with constant coefficients. This has been analysed for a long time and powerful algorithms have been developed and are available to handle them. Therefore when the size of the model increases it is very efficient to use the above algorithms which are implemented in various computer software packages.

It has to be noted that the size of the graph increases exponentially with the number of components: this is the so-called combinatorial explosion. Therefore this technique is usable without approximations only for small systems.

When non-exponential laws have to be handled -semi-Markov models- then Monte Carlo simulation (see B.6.6.8) should be used.

References:

IEC 61165:2006, *Application of Markov techniques*

The Theory of Stochastic Processes. R. E. Cox and H. D. Miller, Methuen and Co. Ltd., London, UK, 1963

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

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

Sécurisation des architectures informatiques. Jean-Louis Boulanger, Hermès – Lavoisier, 2009, ISBN: 978-2-7462-1991-5

B.6.6.7 Reliability block diagrams (RBD)

NOTE 1 This technique/measure is used in Annex B of IEC 61508-6.

NOTE 2 See also C.6.4 “Reliability block diagrams”.

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. It is more a method of representation than a method of analysis.

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.

A RBD is a structural representation of the modelled system. It is a kind of electrical circuit: when the current find a path from the input to the output, that means that the modelled system is working properly, when the circuit is cut that means that the modelled system is failed. This lead to the concept of minimal cut sets which represent the combinations of failures (i.e. places where the RBD is "cut") leading to the failure of the modelled system.

Mathematically a RBD is similar to a fault tree. It represents the logical function linking the states of the individual components (failed or working) to the state of the whole system (failed or working). Therefore the calculations are similar as those described for fault trees.

References:

IEC 61078:2006, *Analysis techniques for dependability – Reliability block diagram and boolean methods*

Sécurisation des architectures informatiques. Jean-Louis Boulanger, Hermès – Lavoisier, 2009, ISBN: 978-2-7462-1991-5

B.6.6.8 Monte-Carlo simulation

NOTE This technique/measure is referenced in Table B.4 of IEC 61508-3 and used in IEC 61508-6 Annex B.

Aim: To simulate real world phenomena by generating random numbers when analytical methods are not practicable.

Description: Monte-Carlo simulations are used to solve two classes of problems:

– probabilistic, where random numbers are used to generate stochastic phenomena; and
– deterministic, which are mathematically translated into an equivalent probabilistic problem (e.g. integral calculations).

The principle of Monte-Carlo simulation is to use random numbers to animate a behavioural functional and dysfunctional model of the system under study. Such behavioural models are provided by state transition models (Markov graph, Petri nets, formal languages, etc.). The Monte-Carlo simulation is run to produce a large statistical sample from which statistical results are obtained.

When using Monte-Carlo simulations care must be taken to ensure that the biases, tolerances or noise have reasonable values. This shall be managed through the confidence interval which is easily obtained from the simulations. Contrary to analytical methods, Monte Carlo simulation is self approximating. Negligible events just not appear without need to identify them to simplify the model.

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.

In the context of this standard, Monte Carlo simulation may be used for SIL calculations and to take into consideration the reliability data uncertainties. With present time computers, SIL4 calculations are easily achieved.

References:

Monte Carlo Methods. J. M. Hammersley, D. C. Handscomb, Chapman & Hall, 1979

Sécurisation des architectures informatiques. Jean-Louis Boulanger, Hermès – Lavoisier, 2009, ISBN: 978-2-7462-1991-5

B.6.6.9 Fault tree models

NOTE 1 See IEC 61508-6 for the use of this technique in the context of analysing hardware safety integrity.

NOTE 2 Fault trees have already been described above as safety validation means (see B.6.6.5). They are also widely used for failure analysis and probabilistic calculations.

Aim: To build by a systematic topdown graphical (effect-cause) approach, the logical function linking the basic events (failure modes) to the top event (unwanted event).

Description: This is both a method of analysis helping the analyst to develop the model step by step and a mathematical model for probabilistic calculations. It allows performing:

– qualitative analysis by identifying and sorting failure scenarios (minimal cut sets or prime implicants),

– semi quantitative analysis by ranking scenarios according to their probabilities of occurrence,

– quantitative analysis by calculating the probability of the top event.

Like RBD (Reliability Block Diagrams), a fault tree represents the logical (boolean) function linking the states of the individual components (failed or working) to the state of the whole system (failed or working). Therefore when the components are independent, probabilistic calculations may be performed just by applying the basic properties of probabilities applied on logical function. This is not so easy because this is a static model which basically works only with genuine (i.e. constant) probabilities). Time dependent probabilities shall be handled cautiously. For example the PFD_{avg} of safety systems comprising periodically proof tested components cannot be calculated straightforwardly and this is even more difficult for *PFH* of safety systems working in continuous mode. Therefore only reliability engineers with a sound understanding of the underlying mathematics should perform un-availability/*PFD* and un-reliability/*PFH* calculations with this method.

Calculations may be done by hand for very simple fault trees but a lot of algorithms have been developed and implemented to handle complex logical equations over the last 50 past years. The state of the art at the present time is using BDD (Binary Decision Diagrams) which is a technique of compact encoding of the logical equation into a computer memory. It is, at the present time, the only method able to perform the probabilistic calculations without approximations on industrial size systems. It is also sufficiently fast to allow handling uncertainties by Monte Carlo simulation.

Reference:

IEC 61025:2006, *Fault tree analysis (FTA)*

B.6.6.10 Generalised Stochastic Petri net models (GSPN)

NOTE 1 See IEC 61508-6 for the use of this technique in the context of analysing hardware safety integrity.

NOTE 2 Petri nets have already been mentioned as semi-formal method (see B.2.3.3). They can also efficiently used in the context of hardware safety integrity.

Aim: To graphically build a functionnal and dysfunctional model behaving as close as possible as the actual modelled system in order to provide an efficient support for Monte carlo simulation.

Description: This is an asynchronous finite state automata as described in B.2.3.3, except that the good property tracked when performing semi-formal validation are no longer existing when modelling the dysfunctionnal behaviour of a safety system. The so-called places (pictured by circles) represent the potental states and the so-called transitions (pictured by rectangles) represents the events likely to occur. In addition of the marking of the places (see B.2.3.3) messages or predicates may be used to validate (enable) the transitions and the delay elapsing from the validation of a transition to its firing may be deterministic or stochastic. This is why those Petri Nets are called "generalised stochastic" Petri nets.

Petri nets constitute flexible behavioural models which prove to be very efficient as Monte Carlo simulation support (see B.6.6.8). Except the accuracy of the Monte Carlo simulation itself which, anyway, is always known, all the limitations of other methods (dependancies,

combinatory explosion, non-exponentail laws, etc.) are overcome. With present time computer this is no longer a problem even for SIL4 evaluations.

References:

IEC 62551, *Analysis techniques for dependability – Petri net modelling (CD1)*[3]

Sécurisation des architectures informatiques. Jean-Louis Boulanger, Hermès – Lavoisier, 2009, ISBN: 978-2-7462-1991-5

B.6.7 Worst-case analysis

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2.

Aim: To avoid systematic failures which arise from unfavourable combinations of the environmental conditions and the component tolerances.

Description: The operational capacity of the system and the component dimensioning is examined or calculated on a theoretical basis. The environmental conditions are changed to their highest permissible marginal values. The most essential responses of the system are inspected and compared with the specification.

B.6.8 Expanded functional testing

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2.

Aim: To reveal failures during the specification and design and development phases. To check the behaviour of the safety-related system in the event of rare or unspecified inputs.

Description: Expanded functional testing reviews the functional behaviour of the safety-related system in response to input conditions which are expected to occur only rarely (for example major failure), or which are outside the specification of the safety-related system (for example incorrect operation). For rare conditions, the observed behaviour of the safety-related system is compared with the specification. Where the response of the safety-related system is not specified, one should check that the plant safety is preserved by the observed response.

References:

Software Testing and Quality Assurance. K. Naik, P. Tripathy, Wiley Interscience, 2008, Print ISBN: 9780471789116 Online ISBN: 9780470382844

The Art of Software Testing, Second Edition. G. Myers et al., Wiley & Sons, New York, 2004, ISBN 0471469122, 9780471469124

Dependability of Critical Computer Systems 3. P. G. Bishop et al,. Elsevier Applied Science, 1990, ISBN 1-85166-544-7

B.6.9 Worst-case testing

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2.

Aim: To test the cases specified during worst-case analysis.

Description: The operational capacity of the system and the component dimensioning is tested under worst-case conditions. The environmental conditions are changed to their highest permissible marginal values. The most essential responses of the system are inspected and compared with the specification.

---

[3] Under consideration.

B.6.10 Fault insertion testing

NOTE This technique/measure is referenced in Tables B.5 and B.6 of IEC 61508-2.

Aim: To introduce or simulate faults in the system hardware and document the response.

Description: This is a qualitative method of assessing dependability. Preferably, detailed functional block, circuit and wiring diagrams are used in order to describe the location and type of fault and how it is introduced; for example: power can be cut from various modules; power, bus or address lines can be open/short-circuited; components or their ports can be opened or shorted; relays can fail to close or open, or do it at the wrong time, etc. Resulting system failures are classified, as in Tables 1 and 2 of IEC 60812, for example. In principle, single steady-state faults are introduced. However, in case a fault is not revealed by the built-in diagnostic tests or otherwise does not become evident, it can be left in the system and the effect of a second fault considered. The number of faults can easily increase to hundreds.

The work is done by a multidisciplinary team and the vendor of the system should be present and consulted. The mean operating time between failure for faults that have grave consequences should be calculated or estimated. If the calculated time is low, modifications should be made.

References:

IEC 60812:2006, *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*

IEC 61069-5:1994, *Industrial-process measurement and control – Evaluation of system properties for the purpose of system assessment – Part 5: Assessment of system dependability*

Annex C
(informative)

Overview of techniques and measures for achieving software safety integrity (see IEC 61508-3)

C.1 General

The overview of techniques contained in this annex should not be regarded as either complete or exhaustive.

C.2 Requirements and detailed design

NOTE Relevant techniques and measures are found in B.2.

C.2.1 Structured diagrammatic methods

NOTE This technique/measure is referenced in Tables A.2 and A.4 of IEC 61508-3.

C.2.1.1 General

Aim: The main aim of structured methods is to promote the quality of software development by focusing attention on the early parts of the lifecycle. The methods aim to achieve this through both precise and intuitive procedures and notations (assisted by computers), to determine and document requirements and implementation features in a logical order and a structured manner.

Description: A range of structured methods exists. Some are designed for traditional data-processing and transaction processing functions, while others are more oriented to process control and real-time applications (which tend to be more safety critical). UML (see C.3.12) contains many examples of structured notations.

Structured methods are essentially "thought tools" for systematically perceiving and partitioning a problem or system. Their main features are the following:

– a logical order of thought, breaking a large problem into manageable stages;

– analysis and documentation of the 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 analysis;

– low intellectual overhead – simple, intuitive, pragmatic;

– often with a strong emphasis on developing a diagrammatic model of the intended system, and CASE tool support for the overall method.

The supporting notations for analysing and documenting problems and system entities (for example processes and data flows) tend to be precise, but notations for expressing the processing functions performed by these entities tend to be more informal. However, some methods do make partial use of (mathematically) formal notations (for example, regular expressions or finite state machines). Increased precision not only reduces the scope for misunderstanding, it provides scope for automatic processing.

Another benefit of structured notations is their visibility, enabling a specification or design to be checked intuitively by a user, against his powerful but unstated knowledge.

This overview describes several structured methods in more detail.

Reference:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

Software Design. D. Budgen, Pearson Education, 2003, ISBN 0201722194, 9780201722192

C.2.1.2 Controlled Requirements Expression (CORE)

Aim: To ensure that all the requirements are determined 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 communication – CORE is designed for requirements expression rather than specification. The approach is structured, and the expression goes through various levels of refinement. The CORE method 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 detected and have to be considered in the subsequent design.

Reference:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

Requirements Engineering. E. Hull, K. Jackson, J. Dick. Springer, 2005, ISBN 1852338792, 9781852338794

C.2.1.3 Jackson System Development (JSD)

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 procedure in which the developer models the real world behaviour 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 specification and design and development 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 discovering 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:

Systems Analysis and Design. D. Yeates, A. Wakefield. Pearson Education, 2003, ISBN 0273655361, 9780273655367

An Overview of JSD. J. R. Cameron. IEEE Transactions on Software Engineering, SE-12, No. 2, February 1986

C.2.1.4 Real-time Yourdon

Aim: The specification and design of real-time systems.

Description: The development scheme underlying this technique assumes a three-stage evolution of a system being developed. The first stage 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 stage involves the actual building of the system in hardware and software. The three stages correspond roughly to the traditional specification and design and development 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 description 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 schemes describing the transformation carried out by the system in response to events and a description of the data the system must hold in order to respond.

The implementation model also divides into submodels, covering the allocation of individual processes to processors and the decomposition of the processes into software modules.

To capture these models, the technique combines a number of other well-known techniques: data-flow diagrams, transformation graphs, 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.

References:

Real-time Systems Development. R. Williams. Butterworth-Heinemann, 2006, ISBN 0750664711, 9780750664714

Structured Development for Real-Time Systems (3 Volumes). P. T. Ward and S. J. Mellor. Yourdon Press, 1985

C.2.2 Data flow diagrams

NOTE This technique/measure is referenced in Tables B.5 and B.7 of IEC 61508-3.

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 have three aspects:

– annotated arrows – represent data flow in and out of the transformation centres, with the annotations documenting what the data is;

– annotated bubbles – represent transformation centres, with the annotation documenting the transformation;

– operators (and, xor) – these operators are used to link the annotated arrows.

Each bubble in a data flow diagram 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 principal advantages is that they show transformations without making any assumptions about how these transformations are implemented. A pure data flow diagram does not include control

information or sequencing information, but this is catered for by real-time extensions to the notation, as in real-time Yourdon (see C.2.1.4).

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 procedure with early attempts refined in stages to produce the final diagram.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

ISO 5807:1985, *Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*

ISO/IEC 8631:1989, *Information technology – Program constructs and conventions for their representation*

C.2.3 Structure diagrams

NOTE This technique/measure is referenced in Table B.5 of IEC 61508-3.

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 modules without including any information about the order of activation of these units. They are drawn using the following four symbols:

- a rectangle annotated with the name of the module;
- a line connecting these rectangles creating structure;
- a circled arrow (circle empty), annotated with the name of data passed to and from elements in the structure chart (normally, the circled arrow is drawn parallel to the line connecting the rectangles in the chart);
- a circled arrow (circle filled), annotated with the name of the control signal passing from one module to another in the structure chart, again the arrow is drawn parallel to the line connecting the two modules.

From any non-trivial data flow diagram, it is possible to derive a number of different structure charts.

Data flow diagrams depict the relationship between information and functions in the system. Structure charts depict the way elements of the system are implemented. Both techniques present valid, though different, views of the system.

References:

Software Design & Development. G. Lancaster. Pascal Press, 2001, ISBN 1741251753, 9781741251753

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

C.2.4 Formal methods

NOTE This technique/measure is referenced in Tables A.1, A.2, A.4 and B.5 of IEC 61508-3.

C.2.4.1 General

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 specification, design or implementation. The resulting description is in a strict notation that 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, because it improves human recognition of the specified behaviour.

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 formal methods are described in the following subclauses of this overview : CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z. Note that other techniques, such as finite state machines and Petri nets (see Annex B), may be considered as formal methods, depending on how strictly the techniques, as used, conform to a rigorous mathematical basis.

References:

Formal Specification: Techniques and Applications. N.Nissanke, Springer-Verlag Telos, 1999, ISBN 1852330023

The Practice of Formal Methods in Safety-Critical Systems. S. Liu, V. Stavridou, B. Dutertre, J. Systems Software 28, 77-87, Elsevier, 1995

Formal Methods: Use and Relevance for the Development of Safety-Critical Systems. L. M. Barroca, J. A. McDermid, The Computer Journal 35 (6), 579-599, 1992

How to Produce Correct Software – An Introduction to Formal Specification and Program Development by Transformations. E. A. Boiten et al, The Computer Journal 35 (6), 547-554, 1992

C.2.4.2 Calculus of Communicating Systems (CCS)

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

Description: 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.

Reference:

Communication and Concurrency. R. Milner. Pearson Education, 1989, ISBN 9780131150072

C.2.4.3 Communicating Sequential Processes (CSP)

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, i.e. a permissible sequence of events).

A system is modelled as a network of independent processes, composed sequentially or in parallel. Each process is described in terms of all of its possible behaviours. 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.

Reference:

Communicating Sequential Processes: The First 25 Years. A. Abdallah, C. Jones, J. Sanders (Eds.). Springer, 2004, ISBN 3540258132, 9783540258131

C.2.4.4 Higher Order Logic (HOL)

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

Description: HOL 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 logic of computable functions (LCF) system.

Reference:

Higher-Order Computational Logic. J. Lloyd. In Computational Logic: Logic Programming and Beyond, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2002, ISBN 978-3-540-43959-2

C.2.4.5 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 aspects of the abstract data type language ACT ONE. The process description aspect of LOTOS could, however, be used with other formalisms for the description of abstract data types.

References:

Model Checking for Software Architectures. R. Mateescu. In Software Architecture, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2004, ISBN 978-3-540-22000-8

ISO 8807:1989, *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*

C.2.4.6 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 equations. 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 (rewriting) 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 used for the specification of both small- and large-scale industrial applications.

References:

Software Engineering with OBJ: Algebraic Specification in Action. J. Goguen, G. Malcolm. Springer, 2000, ISBN 0792377575, 9780792377573

C.2.4.7 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 (for example "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 element 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 description.

Reference:

Mathematical Logic for Computer Science. M. Ben-Ari. Springer, 2001, ISBN 1852333197, 9781852333195

C.2.4.8 VDM, VDM++ – Vienna Development Method

Aim: The systematic specification and implementation of sequential (VDM) and concurrent real-time (VDM++) 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 described 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 the 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 determined 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 sequentially structured programs or the sequential processes in concurrent systems.

The object-oriented and concurrent real-time extension of VDM, VDM++, is a formal specification language based on the ISO language VDM-SL and on the object-oriented language Smalltalk.

VDM++ provides a wide range of constructs such that a user can formally specify concurrent real-time systems in an object-oriented fashion. In VDM++ a complete formal specification consists of a collection of class specifications and optionally a workspace.

Real-time provisions of VDM++ are:

- temporal expressions are provided to denote both the current moment and the method invocation moment within a method body;
- a timed post expression can be added to a method to specify the upper (or lower) bounds of the execution time for correct implementations;
- time continuous variables have been introduced. With assumption and effect clauses one can specify relations (for example differential equations) between these functions of time. This feature has proven to be very useful in the specification of requirements of systems which operate in a time continuous environment. Refinement steps lead to discrete software solutions for these kinds of systems.

References:

ISO/IEC 13817-1:1996, *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*

Systematic Software Development using VDM. C. B. Jones. Prentice-Hall. 2nd Edition, 1990

Conformity Clause for VDM-SL, G. I. Parkin and B. A. Wichmann, Lecture Notes in Computer Science 670, FME'93 Industrial-Strength Formal Methods, First International Symposium of Formal Methods in Europe. Editors: J. C. P. Woodcock and P. G. Larsen. Springer Verlag, 501-520

C.2.4.9 Z

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.

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 described 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:

Formal Specification using Z, 2nd Edition. D. Lightfoot. Palgrave Macmillan, 2000, ISBN 9780333763278

The B-Method. S. Schneider. Palgrave Macmillan, 2001, ISBN 9780333792841

C.2.5 Defensive programming

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-3.

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.

There are two overlapping areas of defensive techniques. Intrinsic error-safe software is designed to accommodate its own design shortcomings. These shortcomings may be due to mistakes in 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.

Fault 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, including 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:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

Dependability of Critical Computer Systems: Guidelines Produced by the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7, Systems Reliability, Safety, and Security). Elsevier Applied Science, 1989, ISBN 1851663819, 9781851663811

C.2.6 Design and coding standards

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-3.

C.2.6.1 General

Aim: To facilitate verifiability, to encourage a team-centred, objective approach 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 comprise the design and development methods to be followed (for example JSP, Petri nets, etc.) and the related coding standards (see C.2.6.2).

These rules are made to allow for ease of development, verification, assessment and maintenance. Therefore they should take into account available tools, in particular analysers and reverse engineering tools.

References:

IEC 60880:2006, *Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions*

Verein Deutscher Ingenieure. Software-Zuverlassigkeit – Grundlagen, Konstruktive Mass-nahmen, Nachweisverfahren. VDI-Verlag, 1993, ISBN 3-18-401185-2

C.2.6.2 Coding standards

NOTE This technique/measure is referenced in Table B.1 of IEC 61508-3.

Aim: To reduce the likelihood of errors in the safety-related code and to facilitate its verification.

Description: The following principles indicate how safety-related coding rules (for any programming language) can assist in complying with the IEC 61508-3 normative requirements and in achieving the informative "desirable properties" (see Annex F). Account should be taken of available support tools.

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Modular approach (Table A.2-7, Table A.4-4)	Software module size limit (Table B.9–1) and software complexity control (Table B.9–2). Examples: • Specification of "local" size and complexity metrics and limits (for modules) • Specification of "global" complexity metrics and limits (for overall modules organisation) • Parameter number limit / fixed number of subprogram parameters (Table B.9–4) Information hiding/encapsulation (Table B.9–3): e.g., incentives for using particular language features. Fully defined interface (Table B.9–6). Examples: • Explicit specification of function signatures • Failure assertion programming (Table A.2-3a) and data verification (7.9.2.7), with explicit specification of pre-conditions and post-conditions for functions, of assertions, of data types invariants
Code understandability • Promote code understandability (7.4.4.13) • Readable, understandable and testable (7.4.6)	Naming conventions promoting meaningful, unambiguous names. Example: avoidance of names that could be confounded (e.g., IO and I0). Symbolic names for numeric values. Procedures and guidelines for source code documentation (7.4.4.13). For example: • Explain why's and meanings (and not only what), • Caveats • Side effects Where practicable, the following information shall be contained in the source code (7.4.4.13): • Legal entity (for example: company, author(s), etc.) • Description • Inputs and outputs • Configuration management history (See also Modular Approach)

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Verifiability and testability • Facilitate verification and testing (7.4.4.13) • Facilitate the detection of design or programming mistakes (7.4.4.10) • Formal verification (Table A.5 - 9) • Formal proof (Table A.9 - 1)	• Wrappers for “critical” library functions, to check pre- and post-conditions • Incentives for using language features that can express restrictions on the use of particular data elements or functions (e.g., const) • For tool supported verification: rules for complying with the limitations of the selected tools (provided this does not impair more essential goals) • Limited use of recursion (Table B.1 – 6) and other forms of circular dependencies (See also Modular Approach)
Static verification of conformance to the specified design (7.9.2.12)	Coding guidelines for the implementation of specific design concepts or constraints. For example: • Coding guidelines for cyclic behaviour, with guaranteed maximum cycle time (Table A.2-13a) • Coding guidelines for time-triggered architecture (Table A.2-13b) • Coding guidelines for event-driven architecture, with guaranteed maximum response time (Table A.2-13c) • Loops with a statically determined maximum number of iterations (except for the infinite loop of the cyclic design) • Coding guidelines for static resource allocation (Table A.2-14) and avoidance of dynamic objects (Table B.1–2) • Coding guidelines for static synchronisation of access to shared resources (Table A.2-15) • Coding guidelines to comply with limited use of interrupts (Table B.1–4) • Coding guidelines to avoid dynamic variables (Table B.1–3a) • Online checking of the installation of dynamic variables (Table B.1–3b) • Coding guidelines to ensure compatibility with other programming languages used (7.4.4.10) Guidelines to facilitate traceability with design

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Language subset (Table A.3 - 3) • Proscribe unsafe language features (7.4.4.13) • Use only defined language features (7.4.4.10) • Structured programming (Table A.4 - 6) • Strongly typed programming language (Table A.3 - 2) • No automatic type conversion (Table B.1 - 8)	Exclusion of language features leading to unstructured designs. E.g., • Limited use of pointers (Table B.1–5) • Limited use of recursion (Table B.1–6) • Limited use of C-like unions • Limited use of Ada or C++-like exceptions • No unstructured control flow in programs in higher level languages (Table B.1–7) • One entry/one exit point in subroutines and functions (Table B.9-5) • No automatic type conversion • Limited use of side effects not apparent from functions signatures (e.g., of static variables). No side effects in evaluation of conditions and all forms of assertions. Limited or documented-only use of compiler-specific features. Limited use of potentially misleading language constructs. Rules to be applied when these language features are used nonetheless.
Good programming practice (7.4.4.13)	When applicable: • Coding guidelines to ensure that, when necessary, floating point expressions are evaluated in the right order (e.g., “a-b+c” is not always equal to “a+c-b”) • In floating point comparisons: use only inequalities (less than, less or equal to, greater than, greater or equal to) instead of strict equality • Guidelines regarding conditional compilation and “pre-processing” • Systematic checking of return conditions (success / failure) Documentation, and, when possible, automation of the production of executable code (makefiles). Avoidance of side effects not apparent from functions signatures. When such side effects exist, guidelines to document them. Bracketing when operators precedence is not absolutely obvious. Catching of supposedly impossible situations (e.g., a “default” case in C “switches”). Use of “wrappers” for critical modules, in particular to check pre- and post-conditions and return conditions. Coding guidelines to comply with known compiler errors and limits set by compiler assessment.

C.2.6.3 No dynamic variables or dynamic objects

NOTE This technique/measure is referenced in Tables A.2 and B.1 of IEC 61508-3.

Aim: To exclude

- unwanted or undetected overlay of memory;
- bottlenecks of resources during (safety-related) run-time.

Description: In the case of this measure, dynamic variables and dynamic objects are those variables and objects that have their memory allocated and absolute addresses determined at run-time. The value of allocated memory and its addresses depend on the state of the system at the moment of allocation, which means that it cannot be checked by the compiler or any other off-line tool.

Because the number of dynamic variables and objects, and the existing free memory space for allocating new dynamic variables or objects, depends on the state of the system at the moment of allocation, it is possible for faults to occur when allocating or using the variables or objects. For example, when the amount of free memory at the location allocated by the system is insufficient, the memory contents of another variable can be inadvertently overwritten. If dynamic variables or objects are not used, these faults are avoided.

Restrictions on the use of dynamic objects are needed where the dynamic behaviour cannot be accurately predicted by some static analysis (i.e. in advance of the program execution), and therefore predictable program execution cannot be guaranteed.

C.2.6.4 On-line checking during creation of dynamic variables or dynamic objects

NOTE 1 This technique/measure is referenced in Table B.1 of IEC 61508-3.

Aim: To check that the memory to be allocated to dynamic variables and objects is free before allocation takes place, ensuring that the allocation of dynamic variables and objects during run-time does not impact existing variables, data or code.

Description: In the case of this measure, dynamic variables are those variables that have their memory allocated and absolute addresses determined at run-time (variables in this sense are also the attributes of object instances).

By means of hardware or software, the memory is checked to ensure it is free before a dynamic variable or object is allocated to it (for example, to avoid stack overflow). If allocation is not allowed (for example if the memory at the determined address is not sufficient), appropriate action must be taken. After a dynamic variable or object has been used (for example, after exiting a subroutine) the whole memory which was allocated to it must be freed.

NOTE 2 An alternative is to demonstrate statically that memory will be adequate in all cases.

C.2.6.5 Limited use of interrupts

NOTE This technique/measure is referenced in Table B.1 of IEC 61508-3.

Aim: To keep software verifiable and testable.

Description: The use of interrupts should be restricted. Interrupts may be used if they simplify the system. Software handling of interrupts should be inhibited during critical parts (for example time critical, critical to data changes) of the executed functions. If interrupts are used, then parts not interruptible should have a specified maximum computation time, so that the maximum time for which an interrupt is inhibited can be calculated. Interrupt usage and masking should be thoroughly documented.

C.2.6.6 Limited use of pointers

NOTE This technique/measure is referenced in Table B.1 of IEC 61508-3.

Aim: To avoid the problems caused by accessing data without first checking range and type of the pointer. To support modular testing and verification of software. To limit the consequence of failures.

Description: In the application software, pointer arithmetic may be used at source code level only if pointer data type and value range (to ensure that the pointer reference is within the correct address space) are checked before access. Inter-task communication of the application software should not be done by direct reference between the tasks. Data exchange should be done via the operating system.

C.2.6.7 Limited use of recursion

NOTE This technique/measure is referenced in Table B.1 of IEC 61508-3.

Aim: To avoid unverifiable and untestable use of subroutine calls.

Description: If recursion is used, there must be a clear criterion which makes predictable the depth of recursion.

C.2.7 Structured programming

NOTE This technique/measure is referenced in Table A.4 of IEC 61508-3.

Aim: To design and implement the program in a way that it is practical to analyse without it being executed. The program may contain only an absolute minimum of statistically untestable behaviour.

Description: The following principles should be applied to minimise structural complexity:

- divide the program into appropriately small software modules, ensuring they are decoupled as far as possible and all interactions are explicit;
- compose the software module control flow using structured constructs, that is sequences, iterations and selection;
- keep the number of possible paths through a software module small, and the relation between the input and output parameters as simple as possible;
- avoid complicated branching and, in particular, avoid unconditional jumps (goto) in higher level languages;
- where possible, relate loop constraints and branching to input parameters;
- avoid using complex calculations as the basis of branching and loop decisions.

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 (for example some safety critical systems).

References:

Concepts in Programming Languages. J. C. Mitchell. Cambridge University Press, 2003, ISBN 0521780985, 9780521780988

A Discipline of Programming. E. W. Dijkstra. Englewood Cliffs NJ, Prentice-Hall, 1976

C.2.8 Information hiding/encapsulation

NOTE This technique/measure is referenced in Table B.9 of IEC 61508-3.

Aim: To prevent unintended access to data or procedures and thereby support a good program structure.

Description: Data that is globally accessible to all software elements can be accidentally or incorrectly modified by any of these elements. 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 (for example 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.

In this connection, the concept of abstract data types should be used. If direct support is not provided, then it may be necessary to check that the abstraction has not been inadvertently broken.

References:

Concepts in Programming Languages. J. C. Mitchell. Cambridge University Press, 2003, ISBN 0521780985, 9780521780988

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

C.2.9 Modular approach

NOTE This technique/measure is referenced in Tables A.4 and B.9 of IEC 61508-3.

Aim: Decomposition of a software system 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 software module (or equivalently, subprogram) should have a single well-defined task or function to fulfil;
- connections between software modules should be limited and strictly defined, coherence in one software module shall be strong;
- collections of subprograms should be built providing several levels of software modules;
- subprogram sizes should be restricted to some specified value, typically two to four screen sizes;
- subprograms should have a single entry and a single exit only;
- software modules should communicate with other software modules via their interfaces – where global or common variables are used they should be well structured, access should be controlled and their use should be justified in each instance;
- all software module interfaces should be fully documented;
- any software module’s interface should contain only those parameters necessary for its function. However, this recommendation is complicated by the possibility that a programming language may permit default parameters, or that an object-oriented approach is used.

References:

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

Concepts in Programming Languages. J. C. Mitchell. Cambridge University Press, 2003, ISBN 0521780985, 9780521780988

C.2.10 Use of trusted/verified software elements

NOTE This technique/measure is referenced in Tables A.2, C.2, A.4 and C.4 of IEC 61508-3.

Aim: To avoid the need for software designs and elements to be extensively revalidated or redesigned for each new application. To take advantage of designs which have not been formally or rigorously verified, but for which considerable operational history is available. To take advantage of a pre-existing software element which has been verified for a different application and for which a body of verification evidence exists.

Description: This measure verifies that the software elements are sufficiently free from systematic design faults and/or operational failures.

It is generally impractical to build a complex system from the most rudimentary parts. It is generally necessary to make use of major subassemblies (“elements”, see IEC 61508-4, 3.4.5 and 3.2.8) that have been previously developed to provide some useful function and which can be reused to implement some part of the new system.

Well-designed and structured PESs are made up of a number of software elements which are clearly distinct and which interact with each other in clearly specified ways. Building up a library of such generally applicable software elements which can be reused in several applications allows much of the resource necessary for validating the designs to be shared by more than one application.

However, for safety related applications it is essential to have sufficient confidence that the new system incorporating these pre-existing elements has the required safety integrity, and that safety is not compromised by some incorrect behaviour of the pre-existing element.

Two viewpoints are possible in order to gain confidence that the behaviour of a pre-existing element can be accurately known:

- to analyse a comprehensive operational history of the element to demonstrate that the element has been “proven-in-use”;

- to assess a body of verification evidence that has been gathered on the behaviour of the element to determine if the element meets the requirements of this standard.

C.2.10.1 Proven-in-use

Only in rare cases will “proven-in-use” (see IEC 61508-4, 3.8.18) be a sufficient argument that a trusted software element achieves the necessary safety integrity. For complex elements with many possible functions (e.g. an operating system), it is essential to establish which functions of the element are actually sufficiently proven-in-use. For example, where a self-test routine is provided to detect faults, if no failure occurs within the operating period, one cannot consider the self-test routine for fault detection as being proven-in-use.

A software element can be considered to be proven-in-use if it fulfils the following criteria:

- unchanged specification;
- systems in different applications;
- at least one year of service history;
- operating time according to the safety integrity level or suitable number of demands; demonstration of a non-safety-related failure rate of less than
  - 10^{-2} per demand (year) with a confidence of 95 % requires 300 operational runs (years),
  - 10^{-5} per demand (year) with a confidence of 99,9 % requires 690 000 operational runs (years);

NOTE 1 See Annex D for some mathematical aspects supporting the above numerical estimates. See also B.5.4 for a similar measure and statistical approach.

- all of the operating experience must relate to a known demand profile of the functions of the software element, to ensure that increased operating experience genuinely leads to an increased knowledge of the behaviour of the software element relative to that demand profile;
- no safety-related failures.

NOTE 2 A failure which may not be safety critical in one context can be safety critical in another, and vice versa.

To enable verification that software element fulfils the criteria, the following must be documented:

- exact identification of each system and its elements, including version numbers (for both software and hardware);
- identification of users, and time of application;
- operating time;
- procedure for the selection of the user-applied systems and application cases;
- procedures for detecting and registering failures, and for removing faults.

C.2.10.2 Assess a body of verification evidence

A pre-existing software element (see IEC 61508-4, 3.2.8) is one that already exists and has not been developed specifically for the current project or SRS. The pre-existing software could be a commercially available product, or it could have been developed by some organisation for a previous product or system. Pre-existing software may or may not have been developed in accordance with the requirements of this standard.

In order to assess the safety integrity of the new system incorporating the pre-existing software, a body of verification evidence is needed to determine the behaviour of the pre-existing element. This may be derived (1) from the element supplier's own documentation and records of the development process of the element, or (2) it may be created or supplemented by additional qualification activities undertaken by the developer of the new safety related system, or by third parties. This is the "Safety Manual for compliant items" that defines the capabilities and limitations of the potentially reusable software element.

In any case, a Safety Manual for compliant items must exist (or must be created) that is adequate to make possible an assessment of the integrity of a specific Safety Function that depends wholly or partly on the reused element. If not, a conservative conclusion must be drawn that the element has not been justified for safety-related reuse. (This is not to say that the element cannot be justified in any case, but simply that insufficient evidence was found in this particular case.)

This standard has specific requirements for the contents of the Safety Manual for compliant items, see IEC 61508-2 Annex D and IEC 61508-3 Annex D, and IEC 61508-3 7.4.2.12 and 7.4.2.13.

As a very brief indication of content, the Safety Manual for compliant items will address the following:

- that the element's design is known and documented;
- the element has been subject to verification and validation using a systematic approach with documented testing and review of all parts of the element's design and code;
- that unused and unneeded functions of the element will not prevent the new system from meeting its safety requirements;
- that all credible failure mechanisms of the element in the new system have been identified and that appropriate mitigation has been implemented.

A functional safety assessment of the new system must establish that the reused element is applied strictly within the limits of capability that have been justified by the evidence and assumptions in the element's Compliant Item Safety Manual.

References:

Component-Based Software Development: Case Studies. Kung-Kiu Lau. World Scientific, 2004, ISBN 9812388281, 9789812388285

Software Reuse and Reverse Engineering in Practice. P. A. V. Hall (ed.), Chapman & Hall, 1992, ISBN 0-412-39980-6

Software criticality analysis of COTS/SOUP. P.Bishop, T.Clement, S.Guerra. In Reliability Engineering & System Safety, Volume 81, Issue 3, September 2003, Elsevier Ltd., 2003

C.2.11 Traceability

Aim: To maintain consistency between lifecycle stages.

Description: In order to ensure that the software that results from lifecycle activities meets the requirements for correct operation of the safety-related system, it is essential to ensure consistency between the lifecycle stages. A key concept here is that of "traceability" between activities. This is essentially an impact analysis to check (1) that decisions made at an earlier stage are adequately implemented in later stages (forward traceability), and (2) that decisions made at a later stage are actually required and mandated by earlier decisions.

Forward traceability is broadly concerned with checking that a requirement is adequately addressed in later lifecycle stages. Forward traceability is valuable at several points in the safety lifecycle:

- from the system safety requirements to the software safety requirements;
- from the Software Safety Requirements Specification, to the software architecture;
- from the Software Safety Requirements Specification, to the software design;
- from the Software Design Specification, to the module and integration test specifications;
- from the system and software design requirements for hardware/software integration, to the hardware/software integration test specifications;
- from the Software Safety Requirements Specification, to the software safety validation plan;
- from the Software Safety Requirements Specification, to the software modification plan (including reverification and revalidation);
- from the Software Design Specification, to the software verification (including data verification) plan;

- from the requirements of IEC 61508-3 Clause 8, to the plan for software functional safety assessment.

Backward traceability is broadly concerned with checking that every implementation (interpreted in a broad context, and not confined to code implementation) decision is clearly justified by some requirement. If this justification is absent, then the implementation contains something unnecessary that will add to the complexity but not necessarily address any genuine requirement of the safety-related system. Backward traceability is valuable at several points in the safety lifecycle:

- from the safety requirements, to the perceived safety needs;
- from the software architecture, to the Software Safety Requirements Specification;
- from the software detailed design to the software architecture;
- from the software code to the software detailed design;
- from the software safety validation plan, to the Software Safety Requirements Specification;
- from the software modification plan, to the Software Safety Requirements Specification;
- from the software verification (including data verification) plan, to the Software Design Specification.

Reference:

Requirements Engineering. E. Hull, K. Jackson, J. Dick. Springer, 2005, ISBN 1852338792, 9781852338794

C.2.12 Stateless software design (or limited state design)

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

Aim: To limit the complexity of software behaviour.

Description: Consider a software program that processes a sequence of transactions: it receives a sequence of inputs and produces an output in response to each input. The program may also memorise some or all of its history in a “state”, and may take this state into account when calculating how to respond to future inputs.

Where the program’s output in response to a specific input is completely determined by that input, then the program is said to be memoryless or stateless. Each input/output transaction is complete in the sense that no transaction is influenced by any earlier transaction, and a specific input always results in the same associated output.

In contrast, where the program takes account of both its specific input and also its memorised state in calculating its output, then the program is capable of more complex behaviour because it can deliver different outputs in response to the same input on different occasions. The response to a specific input may depend on the context (i.e. the previous inputs and outputs) in which the input is received. A further consideration that is relevant to some applications (typically communications) is that the program’s behaviour can be particularly sensitive to changes in the stored state, whether inadvertently or maliciously introduced.

Stateless (or limited state) design is a general approach that aims to minimise the potential complexity of software behaviour by avoiding or minimising the use of state information in the software design.

References:

Introduction to Automata Theory, Languages, and Computation (3rd Edition). J. Hopcroft, R. Motwani, J. Ullman, Addison-Wesley Longman Publishing Co, 2006, ISBN:0321462254

Stateless connections. T. Aura, P Nikander. In Proc International Conference on Information and Communications Security (ICICS'97), ed Yongfei Han. Springer, 1997, ISBN 354063696X, 9783540636960

C.2.13 Offline numerical analysis

NOTE This technique/measure is referenced in Table A.9 of IEC 61508-3.

Aim: To ensure the accuracy of numerical calculations.

Description: Numerical inaccuracy may arise in the calculation of a mathematical function as a consequence of using finite representations of ideal functions and numbers. Truncation error is introduced when a function is approximated by a finite number of terms of an infinite series such as a Fourier series. Rounding error is introduced by the finitely accurate representation of real numbers in a physical computer. When anything but the simplest calculation is performed in floating point, the validity of the calculation must be checked to ensure that the accuracy required by the application is actually achieved.

Reference:

Guide to Scientific Computing. P.R. Turner. CRC Press, 2001, ISBN 0849312426, 9780849312427

C.2.14 Message sequence charts

NOTE This technique/measure is referenced in Tables B.7 and C.17 of IEC 61508-3.

Aim: To assist the capture of system requirements in the early design stages of software development including requirements and software architectural design. In UML, the name "System Sequence Diagram" is used for this notation.

Description: The Message Sequence Chart is a diagrammatic mechanism for describing the behaviour of a system in terms of the communication that takes place between the system actors (and actor may be to a human being, a computer system, or a software element or object, depending on the design phase). For each actor, a vertical "lifeline" is drawn on the diagram and arrows between the lifelines are used to represent messages. Actions upon receipt of messages can be optionally shown on the diagrams as boxes. A collection of scenarios (describing both desirable and undesirable behaviour) is built up as a specification of the required system behaviour. These scenarios have several uses. They can be animated to demonstrate the system behaviour to end-users. They can be transformed into an executable implementation of the system. They can form the basis of test data.

UML contains extensions to the orginal concept of the Message Squence Chart in the form of selection and iteration constructs which allow scenarios to branch and loop, providing a more compact notation. Sub-diagrams can also be defined which can be referenced from a number of higher level sequence diagrams. Timer and external events can also be represented.

References:

"*Message Sequence charts*", D. Harel, P. Thiagarajan. In UML for Real: Design of Embedded Real-Time Systems. ed. L. Lavagno. Springer, 2003, ISBN 1402075014, 9781402075018

ISO/IEC 19501:2005, *Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2*

C.3 Architecture design

C.3.1 Fault detection and diagnosis

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

Aim: To detect faults in a system, which might lead to a failure, thus providing the basis for counter-measures in order to minimise the consequences of failures.

Description: Fault detection is the activity of checking a system for erroneous states (caused 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 acts in combination with parallel elements, relinquishing control when it detects its own results are incorrect, is called self-checking.

Fault detection is based on the principles of redundancy (mainly to detect hardware faults – see IEC 61508-2 Annex A) 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 diverse monitor technique; and for hardware: introducing additional 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 physical (temperature, voltage etc), logical (error detecting codes), functional (assertions) or external (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 should be applied at the smallest subsystem level, since smaller subsystems allow a more detailed diagnosis of faults (detection of erroneous states).

Integrated enterprise-wide information systems can routinely communicate the status of safety systems, including diagnostic testing information, to other supervisory systems. If an anomaly is detected, it can be highlighted and used to trigger corrective action before a hazardous situation develops. Lastly, if an incident does occur, documentation of such anomalies can aid the subsequent investigation.

Reference: *Dependability of Critical Computer Systems 1*. F. J. Redmill, Elsevier Applied Science, 1988, ISBN 1-85166-203-0

C.3.2 Error detecting and correcting codes

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

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 r errors to be detected and corrected. Two example types are Hamming codes and polynomial codes.

It should be noted that in safety-related systems it will normally be necessary to discard faulty data rather than try to correct it, since only a predetermined fraction of errors may be corrected properly.

Reference:

Fundamentals of Error-correcting Codes, W. Huffman, V. Pless. Cambridge University Press, 2003, ISBN 0521782805, 9780521782807

C.3.3 Failure assertion programming

NOTE This technique/measure is referenced in Table A.17 of IEC 61508-2, and Tables A.2 and C.2 of IEC 61508-3.

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 reports the error.

For example,

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

References:

Exploiting Traces in Program Analysis. A. Groce, R. Joshi. Lecture Notes in Computer Science vol 3920, Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-33056-1

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

C.3.4 Diverse monitor

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

Aim: To protect against residual specification and implementation faults in software which adversely affect safety.

Description: Two monitoring approaches can be distinguished: (1) the monitor and the monitored function in the same computer, with some guarantee of independence between them; and (2) the monitor and the monitored function in separate computers.

A diverse monitor is an external monitor, implemented on an independent computer to a different specification. This diverse monitor is solely concerned with ensuring that the main computer performs safe, not necessarily correct, actions. The diverse monitor continuously monitors the main computer. The diverse monitor 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 diverse monitor or the main computer.

Hardware and software of the diverse monitor should be classified and qualified according to the appropriate SIL.

Reference:

Requirements based Monitors for Real-Time Systems, D. Peters, D. Parnas. IEEE Transactions on Software Engineering, vol. 28, no. 2, 2002

C.3.5 Software diversity (diverse programming)

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

Aim: Detect and mask residual software design and implementation 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 is designed and 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.

An essential requirement is that the *N* versions be independent of each other in some sense, so that they do not all fail simultaneously due to the same cause. In practice it may be very difficult to achieve and to demonstrate the version independence that is the foundation of the *N*-version approach.

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, as follows.

– If the system has a safe state, then it is feasible to demand complete agreement (all *N* agree) otherwise an output value is used that will cause the system to reach the safe state. 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, nor does it avoid errors in the interpretation of the specification, but it provides a measure to detect and mask before they can affect safety.

References:

Modelling software design diversity – a review, B. Littlewood, P. Popov, L. Strigini. ACM Computing Surveys, vol 33, no 2, 2001

The N-Version Approach to Fault-Tolerant Software, A.Avizienis, IEEE Transactions on Software Engineering, vol. SE-11, no. 12 pp.1491-1501, 1985

An experimental evaluation of the assumption of independence in multi-version programming, J.C. Knight, N.G. Leveson. IEEE Transactions on Software Engineering, vol. SE-12, no 1, 1986

In Search of Effective Diversity: a Six Language Study of Fault-Tolerant Flight Control Software. A. Avizienis, M. R. Lyu and W. Schutz. 18th Symposium on Fault-Tolerant Computing, Tokyo, Japan, 27-30 June 1988, IEEE Computer Society Press, 1988, ISBN 0-8186-0867-6

C.3.6 Backward recovery

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

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 incrementally (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.

References:

Looking into Compensable Transactions. Jing Li, Huibiao Zhu, Geguang Pu, Jifeng He. In Software Engineering Workshop, 2007. SEW 2007. IEEE, 2007, ISBN 978-0-7695-2862-5

Software Fault Tolerance (Trends in Software, No. 3), M. R. Lyu (ed.), John Wiley & Sons, April 1995, ISBN 0471950688

C.3.7 Re-try fault recovery mechanisms

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

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 reboot and a re-start procedure or a small re-scheduling and re-starting task, after a software time-out or a task monitoring 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 (checksum, etc.) or from a communication acknowledgement response time-out.

Reference:

Reliable Computer Systems: Design and Evaluation, D.P. Siewiorek, R.S. Schwartz. A.K. Peters Ltd., 1998, ISBN 156881092X, 9781568810928

C.3.8 Graceful degradation

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

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 ensures that if there is insufficient resources to carry out all the system functions, 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, in which case 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 including software. It must be taken into account from the topmost design phase.

References:

Towards the Integration of Fault, Resource, and Power Management, T. Siridakis. In Computer Safety, Reliability, and Security: 23rd International Conference, SAFECOMP 2004. Eds. Maritta Heisel et. al. Springer, 2004, ISBN 3540231765, 9783540231769

Achieving Critical System Survivability Through Software Architectures, J.C. Knight, E.A. Strunk. Springer Berlin / Heidelberg, 2004, 978-ISBN 3-540-23168-4

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, 1987, ISBN 3-211-81941-X

Fault Tolerance, Principle and Practices. T. Anderson and P. A. Lee, Vol. 3 of Dependable Computing and Fault-Tolerant Systems, Springer Verlag, 1987, ISBN 3-211-82077-9

C.3.9 Artificial intelligence fault correction

NOTE 1 This technique/measure is referenced in Table A.2 of IEC 61508-3.

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

Description: Fault forecasting (calculating trends), fault correction, maintenance and supervisory actions may be supported by artificial intelligence (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 in such a way that faults may be corrected and the effects of failures be minimised, in order to meet the desired safety integrity.

NOTE 2 See C.3.2 for warning about correcting faulty data, and item 5, Table A.2 of IEC 61508-3 for negative recommendations concerning this technique.

Reference:

Fault Diagnosis: Models, Artificial Intelligence, Applications. J. Korbicz, J. Koscielny, Z. Kowalczuk, W. Cholewa. Springer, 2004, ISBN 3540407677, 9783540407676

C.3.10 Dynamic reconfiguration

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

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 be of little importance.

This technique must be considered at the first system design stage.

Reference:

Dynamic Reconfiguration of Software Architectures Through Aspects. C. Costa et al. Lecture Notes in Computer Science, Volume 4758/2007, Springer Berlin / Heidelberg, 2007, ISBN 978-3-540-75131-1

C.3.11 Safety and Performance in real time: Time-Triggered Architecture

NOTE 1 This technique/measure is referenced in Table A.2 of IEC 61508-3.

Aim: Composability and transparent implementation of fault-tolerance into safety-critical real-time systems with predictable behaviour.

Description: In a Time-Triggered Architecture (TTA) system, all system activities are initiated and based on the progression of a globally synchronised time-base. Each application is assigned a fixed time slot on the time-triggered bus, which contains the messages exchanged between the jobs of each application which can therefore be exchanged only according to a defined schedule. In event-driven systems, system activities are triggered by arbitrary events at unpredictable points in time. The key advantages of a TTA are (see reference Scheidler, Heiner et. al.):

– composability, which greatly reduces the effort required for testing and certifying the system;
– transparent implementation of fault-tolerance, which makes the architecture highly recommendable for safety-critical applications;
– provision of a globally synchronised time-base, which facilitates the design of distributed real-time systems.

Communication between nodes is done using the *Time-Triggered Protocol TTP/C* (see reference Kopetz, Hexel et. al.) according to a static schedule, deciding when to transmit a message and whether a received message is relevant for the particular electronic module or not. Access to the bus is controlled by a cyclic *time-division multiple access (TDMA)* schema derived from the global notion of time.

The TTP/C protocol guarantees (see reference Rushby) four basic services (core services) in a network of TTA nodes (see reference Kopetz, Bauer):

– Deterministic and timely message transport: Transport of messages from the output port of the sending element to the input ports of the receiving elements within an a priori known time bound. A fault-tolerant transport service is offered by a time-triggered communication service that is available via the temporal firewall interface which eliminates control error propagation by design and minimises coupling between elements. The timely transport of messages with minimal latency and jitter is crucial for the achievement of control stability in real-time applications.
– Fault-tolerant Clock Synchronization: The communication controller generates a fault-tolerant synchronised global time base (with a precision within a few clock tics) that is provided to the host subsystem.
– Consistent Diagnosis of Failing Nodes (Membership Service): The communication controller informs every SRU (“smallest replaceable unit”) about the state of every other SRU in a cluster with a latency of less than one TDMA round.
– Strong Fault Isolation: A maliciously faulty host subsystem (including its software) can produce erroneous data outputs, but can never interfere in any other way with the correct operation of the rest of a TTP/C cluster. Fail silence in the temporal domain is guaranteed by the time-triggered behaviour of the communication controller.

NOTE 2 Other time-triggered protocols are FlexRay and TT-Ethernet (time-triggered Ethernet).

References:

Time-Triggered Architecture (TTA). C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, C. Temple. In Advances in Information Technologies: The Business Challenge, ed. J-Y. Roger. IOS Press, 1998, ISBN 9051993854, 9789051993851

A Synchronisation Strategy for a TTP/C Controller. H. Kopetz, R. Hexel, A. Krueger, D. Millinger, A. Schedl. SAE paper 960120, Application of Multiplexing Technology SP 1137, Detroit, SAE Press, Warrendale, 1996

The Time-Triggered Architecture. H. Kopetz, G. Bauer. Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software, October 2002

An Overview of Formal Verification for the Time-Triggered Architecture. J. Rushby: Invited paper, Oldenburg, Germany, September 9-12, 2002. Proceedings FTRTFT 2002, Springer LNCS 2469, 2002, ISBN 978-3-540-44165-6

C.3.12 UML

NOTE This technique/measure is referenced in Table B.7 of IEC 61508-3.

Aim: To provide a comprehensive set of notations for modelling the desired behaviour of complex systems

Description: UML is, as the name implies, a collection of requirements and design notations which is intended to provide comprehensive support for software development. Some parts of UML are based upon notations first introduced in other methods (such as system sequence diagrams and state transition diagrams) and other notations are unique to UML. UML is strongly biased towards object-oriented concepts although some of the notations can be used without any necessity to proceed to an object-oriented programming. UML is supported by a number of commercially available CASE tools, many of which are capable of automatically generating code from the UML models.

The UML notations which are most generally applicable to the specification and design of safety related systems are the following:

- Class diagrams
- Use cases
- Activity diagrams
- State transition diagrams (Statecharts)
- System sequence diagrams

Other UML notations are relevant to the expression of software architectural design (software structure) but are not listed specifically here.

State transition diagrams are described in B.2.3.2 and system sequence diagrams in C.2.14. The other notations are described in the following three subsections.

C.3.12.1 Class diagrams

Class diagrams define the classes of objects with which the software has to deal. They are based upon earlier entity-relationship-attribute diagrams but are adapted for object oriented design. Each class (of which there will be one or more instances known as objects at run time) is represented as a rectangle and the various relationships between the classes are shown as lines or arrows. The operations or methods offered by each class, and the data attributes of each class, can be added to the diagram. The relationships which can be represented consist of both reference relationships with their cardinality (an instance of class

A may refer to one or many instances of class B) and specialisation relationships (Class X is a refinement of class Y) with possibly additional methods and attributes. Multiple inheritance can be depicted.

C.3.12.2 Use cases

Use cases provide a textual description of the desired behaviour of the system in reponse to a particular scenario, usually from the point of view of external actors including human users of the system and external systems. Alternative sub-scenarios within a given use case can be used to represent optional behaviour, especially in error response cases. A collection of use cases is developed to provide a sufficiently complete specification of the system requirements. Use clauses can be the starting point for the development more rigorous models such as system sequence diagrams and activity diagrams.

Use case diagrams provide a pictorial representation of the system and the actors who are involved in the use cases, but are not rigorous and only the text of the use case is important for specification.

C.3.12.3 Activity diagrams

An activity diagram shows the intended sequence of actions carried out by a software element (often an object in an object-oriented design) including sequential and iterative behaviour (some aspects look remarkably like a flowchart). Activity diagrams however allow the actions of a number of elements to be described in parallel, with the interactions between the elements shown by arrows on the diagram. Synchronisation points where an activity must wait for one or more inputs from other activities before it can proceed are shown by a symbol similar to a Petri net node.

Reference:

ISO/IEC 19501:2005, *Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2*

C.4 Development tools and programming languages

C.4.1 Strongly typed programming languages

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-3.

Aim: Reduce the probability of faults by using a language which permits a high level of checking by the compiler.

Description: When a strongly typed programming language is compiled, many checks are made on how variable types are used, for example in procedure calls and external data access. Compilation will fail and an error message be produced for any usage that does not conform to predefined rules.

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 type. 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 software modules.

Strongly typed languages usually support other aspects of good software engineering practice such as easily analysable control structures (for example if.. then.. else, do.. while, etc.) which lead to well-structured programs.

Typical examples of strongly typed languages are Pascal, Ada and Modula 2.

Reference:

Concepts in Programming Languages. J. C. Mitchell. Cambridge University Press, 2003, ISBN 0521780985, 9780521780988

C.4.2 Language subsets

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-3.

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

Description: The language is examined to determine 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.

References:

Practical Experiences of Safety- and Security-Critical Technologies, P. Amey, A.J. Hilton. Ada User Journal, June, 2004

Safer C: Developing Software for High-integrity and Safety-critical Systems. L. Hatton, McGraw-Hill, 1994, ISBN 0077076400, 9780077076405

Requirements for programming languages in safety and security software standard. B. A. Wichmann. Computer Standards and Interfaces. Vol. 14, pp 433-441, 1992

C.4.3 Certified tools and certified translators

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-3.

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: 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 (specification, 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.

It is important to note that certified tools and certified translators are usually certified only against their respective language or process standards. They are usually not certified in any way with respect to safety.

References:

The certification of software tools with respect to software standards, P. Bunyakiati et al. In IEEE International Conference on Information Reuse and Integration, IRI 2007, IEEE, 2007, ISBN 1-4244-1500-4

Certified Testing of C Compilers for Embedded Systems. O. Morgan. In: 3rd Institution of Engineering and Technology Conference on Automotive Electronics. IEEE, 2007, ISBN 978-0-86341-815-0

The Ada Conformity Assessment Test Suite (ACATS), version 2.5, Ada Conformity Assessment Authority, http://www. ic.org/compilerstesting.html , Apr. 2002

C.4.4 Tools and translators: increased confidence from use

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-3.

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, where there has been no evidence of improper performance over many prior projects. Translators without operating experience or with any serious known faults should be avoided unless there is some other assurance of correct performance (for example, see C.4.4.1).

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 exists to prove the correctness for all tool or translator parts.

C.4.4.1 Comparison of source program and executable code

Aim: To check that the tools used to produce a PROM image have not introduced any errors into the PROM image.

Description: The PROM image is reverse-engineered to obtain the constituent "object" modules. These "object" modules are reverse-engineered into assembly language files. Using suitable techniques the reverse generated assembly language files are compared with the actual source files originally used to produce the PROM.

The major advantage of the technique is that the tools (compilers, linkers etc.) used to produce the PROM image do not have to be validated for all programs. The technique verifies that source file used for the particular safety-related system are correctly transformed.

References:

Demonstrating Equivalence of Source Code and PROM Contents. D. J. Pavey and L. A. Winsborrow. The Computer Journal Vol. 36, No. 7, 1993

Formal demonstration of equivalence of source code and PROM contents: an industrial example. D. J. Pavey and L. A. Winsborrow. Mathematics of Dependable Systems, Ed. C. Mitchell and V. Stavridou, Clarendon Press, 1995, ISBN 0-198534-91-4

Assuring Correctness in a Safety Critical Software Application. L. A. Winsborrow and D. J. Pavey. High Integrity Systems, Vol. 1, No. 5, pp 453-459, 1996

C.4.5 Suitable programming languages

NOTE This technique/measure is referenced in Table A.3 of IEC 61508-3.

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 an 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-orientated rather than processor/platform machine-orientated. 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; and
- run-time type and array bound checking.

The language should encourage

- the use of small and manageable software modules;
- restriction of access to data in specific software modules;
- definition of variable subranges; and
- any other type of error-limiting constructs.

If safe operation of the system is dependent upon real-time constraints, then the language should also provide for exception/interrupt handling.

It is desirable that the language is supported by a suitable translator, appropriate libraries of pre-existing software modules, a debugger and tools for both version control and development.

Currently, at the time of developing this standard, it is not clear whether object-oriented languages are to be preferred to other conventional ones.

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 processor/platform machine-orientated nature.

A desirable language property is that its design and use should result in programs whose execution is predictable. Given a suitably defined programming language, there is a subset which ensures that program execution is predictable. This subset cannot (in general) be statically determined, although many static constraints may assist in ensuring predictable execution. This would typically require a demonstration that array indices are within bounds, and that numeric overflow cannot arise, etc.

Table C.1 gives recommendations for specific programming languages.

References:

Concepts in Programming Languages. J. C. Mitchell. Cambridge University Press, 2003, ISBN 0521780985, 9780521780988

IEC 60880:2006, *Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions*

IEC 61131-3:2003, *Programmable controllers – Part 3: Programming languages*

ISO/IEC 1539-1:2004, *Information technology – Programming languages – Fortran – Part 1:Base language*

ISO/IEC 7185:1990, *Information technology – Programming languages – Pascal*

ISO/IEC 8652:1995, *Information technology – Programming languages – Ada*

ISO/IEC 9899:1999, *Programming languages – C*

ISO/IEC 10206:1991, *Information technology – Programming languages – Extended Pascal*

ISO/IEC 10514-1:1996, *Information technology – Programming languages – Part 1: Modula-2, Base Language*

ISO/IEC 10514-3:1998, *Information technology – Programming languages – Part 3: Object Oriented Modula-2*

ISO/IEC 14882:2003, *Programming languages – C++*

ISO/IEC/TR 15942:2000, *Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems*

Table C.1 – Recommendations for specific programming languages

	Programming language	SIL1	SIL2	SIL3	SIL4
1	ADA	HR	HR	R	R
2	ADA with subset	HR	HR	HR	HR
3	Java	NR	NR	NR	NR
4	Java with subset (including either no garbage collection or garbage collection which will not cause the application code to stop for a significant period of time). See Annex G for guidance on use of object oriented facilities.	R	R	NR	NR
5	PASCAL (see Note 1)	HR	HR	R	R
6	PASCAL with subset	HR	HR	HR	HR
7	FORTRAN 77	R	R	R	R
8	FORTRAN 77 with subset	HR	HR	HR	HR
9	C	R	–	NR	NR
10	C with subset and coding standard, and use of static analysis tools	HR	HR	HR	HR
11	C++ (see Annex G for guidance on use of object oriented facilities)	R	–	NR	NR
12	C++ with subset and coding standard, and use of static analysis tools (see Annex G for guidance on use of object oriented facilities)	HR	HR	HR	HR
13	Assembler	R	R	–	–
14	Assembler with subset and coding standard	R	R	R	R
15	Ladder diagrams	R	R	R	R
16	Ladder diagram with defined subset of language	HR	HR	HR	HR

	Programming language	SIL1	SIL2	SIL3	SIL4
17	Functional block diagram	R	R	R	R
18	Function block diagram with defined subset of language	HR	HR	HR	HR
19	Structured text	R	R	R	R
20	Structured text with defined subset of language	HR	HR	HR	HR
21	Sequential function chart	R	R	R	R
22	Sequential function chart with defined subset of language	HR	HR	HR	HR
23	Instruction list	R	–	NR	NR
24	Instruction list with defined subset of language	HR	R	R	R

NOTE 1 The recommendationsHR, R and – NR are explained in Annex A of IEC 61508-3.

NOTE 2 System software includes the operating system, drivers, embedded functions and software modules provided as part of the system. The software is typically provided by the safety system vendor. The language subset should be carefully selected to avoid complex structures which may result in implementation faults. Checks should be performed to check for proper use of the language subset.

NOTE 3 The application software is the software developed for a specific safety application. In many cases this software is developed by the end user or by an application oriented contractor. Where a number of programming languages have the same recommendation, the developer should select one which is commonly used by personnel in the industry or facility. The language subset should be carefully selected to avoid complex structures which may result in implementation faults. Checks should be performed to check for proper use of the language subset.

NOTE 4 If a specific language is not listed in the table, it must not be assumed that it is excluded. It should conform to this International Standard.

NOTE 5 There are a number of extensions to the Pascal language including Free Pascal. References to Pascal include these extensions.

NOTE 6 Java is designed to have a run-time garbage collector. A subset of Java can be defined which does not require garbage collection. Some Java implementations provide progressive garbage collection which recovers free memory as the program executes and prevent execution stopping for a period when available memory is exhausted. Hard real time applications should not use any form of garbage collection.

NOTE 7 If the Java implementation requires a run-time interpreter of Java intermediate code, then the interpreter must be treated as part of the safety related software and treated in accordance with the requirements of IEC 61508-3.

NOTE 8 For entries 15-24, see IEC 61131-3.

C.4.6 Automatic software generation

NOTE This technique/measure is referenced in Table A.2 of IEC 61508-3.

Aim: To automate the more error-prone tasks of software implementation.

Description: The system design is described by a model (an executable specification) at a higher level of abstraction than the traditional executable code. The model is transformed automatically by a code generator into executable form. The aim is to improve software quality by eliminating the error-prone manual tasks of coding. A further potential benefit is that more complex designs can be undertaken at the higher abstract level.

References:

Embedded Software Generation from System Level Design Languages, H Yu, R. Domer, D. Gajski. In “ASP-DAC 2004: Proceedings of the ASP-Dac 2004 Asia and South Pacific Design Automation Conference, 2004”, IEEE Circuits and Systems Society. IEEE, 2004, ISBN 0780381750, 9780780381759

Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap?. M.F. van Amstel et. al. In Theory and Practice of Model Transformations: First International Conference, ICMT”. ed. A. Vallecillo. Springer, 2008, ISBN 3540699260, 9783540699262

C.4.7 Test management and automation tools

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-3.

Aim: To encourage a systematic and thorough approach to software and system testing.

Description: The use of appropriate support tools mechanises the more labour-intensive and error-prone tasks in system development and brings the capability for a systematic approach to test management. The availability of support encourages a more thorough approach to both normal and regression testing.

Reference:

Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing. R.Black, John Wiley and Sons, 2002, ISBN 0471223980, 9780471223986

C.5 Verification and modification

C.5.1 Probabilistic testing

NOTE This technique/measure is referenced in Tables A.5, C.15, A.7 and C.17 of IEC 61508-3.

Aim: To gain a quantitative figure about the reliability properties of the investigated software.

Description: The method produces a statistical estimate of software reliability. This quantitative figure may take into account the related levels of confidence and significance and can give

– a failure probability per demand;
– a failure probability during a certain period of time; and
– 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.

Probabilistic considerations are either based on a probabilistic test or on operating experience. Usually, the number of test cases or observed operating cases is very large. Typically, the testing of the demand mode of operation involves considerably less elapsed time than the continuous mode of operation.

Automated testing tools are normally employed to provide test data and supervise test outputs. Large tests are run on large host computers with the appropriate process simulation periphery. Test data is selected both according to systematic and random hardware viewpoints. The overall test control, for example, guarantees a test data profile, while random selection can govern 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 by the mathematical prerequisites that must be fulfilled if the test evaluation is to meet its 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.

In practice, it is very difficult to demonstrate ultra-high levels of reliability using these techniques.

References:

A discussion of statistical testing on a safety-related application. S Kuball, J H R May, Proc IMechE Vol. 221 Part O: J. Risk and Reliability, Institution of Mechanical Engineers, 2007

Estimating the Probability of Failure when Testing Reveals No Failures, W.K. Miller, L.J. Morell, et al.. IEEE Transactions on Software Engineering, VOl. 18, NO.1, pp33-43, January 1992

Reliability estimation from appropriate testing of plant protection software, J. May, G. Hughes, A.D. Lunn. IEE Software Engineering Journal, v10 n6 pp 206-218, Nov 1995 (ISSN: 0268-6961)

Validation of ultra high dependability for software based systems, B. Littlewood and L. Strigini. Comm. ACM 36 (11), 69-80, 1993

C.5.2 Data recording and analysis

NOTE This technique/measure is referenced in Tables A.5 and A.8 of IEC 61508-3.

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

Description: Detailed documentation is maintained during a project, which could include

– testing performed on each software module;

– decisions and their rationale;

– problems and their solutions.

During and at the conclusion of the project this documentation 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.

Reference:

Dependability of Critical Computer Systems 2. F. J. Redmill, Elsevier Applied Science, 1989, ISBN ISBN 1851663819, 9781851663811

C.5.3 Interface testing

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-3.

Aim: To detect errors in the interfaces of subprograms.

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

– all interface variables at their extreme values;

- all interface variables 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 will only be feasible for small interfaces);
- the specified test conditions relevant to each call of each subroutine.

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

C.5.4 Boundary value analysis

NOTE This technique/measure is referenced in Tables B.2, B.3 and B.8 of IEC 61508-3.

Aim: To detect software errors occurring at parameter limits or boundaries.

Description: The input domain of the program is divided into a number of input classes according to the equivalence relation (see C.5.7). 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;
- full 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 the output to exceed the specification boundary values.

If the 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, one and two elements.

References:

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

C.5.5 Error guessing

NOTE This technique/measure is referenced in Tables B.2 and B.8 of IEC 61508-3.

Aim: To remove common programming mistakes.

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. For example: can the buttons be pushed on the front-panel too fast or too often? What happens if two buttons are pushed simultaneously?

Reference:

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

C.5.6 Error seeding

NOTE This technique/measure is referenced in Table B.2 of IEC 61508-3.

Aim: To ascertain whether a set of test cases is adequate.

Description: Some known types of mistake are inserted (seeded) into 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.

$$\frac{\text{Found seeded errors}}{\text{Total number of seeded errors}} = \frac{\text{Found real errors}}{\text{Total number of real errors}}$$

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 types of mistake as well as the seeding positions must reflect the statistical distribution of real errors.

References:

Software Fault Injection: Inoculating Programs Against Errors. J. Voas, G. McGraw. Wiley Computer Pub., 1998, ISBN 0471183814, 9780471183815

Faults, Injection Methods, and Fault Attacks. Chong Hee Kim, Jean-Jacques Quisquater, IEEE Design and Test of Computers, vol. 24, no. 6, pp. 544-545, Nov., 2007

Fault seeding for software reliability model validation. A. Pasquini, E. De Agostino. Control Engineering Practice, Volume 3, Issue 7, July 1995. Elsevier Science Ltd

C.5.7 Equivalence classes and input partition testing

NOTE This technique/measure is referenced in Tables B.2 and B.3 of IEC 61508-3.

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 the partitions previously specified. At least one test case is taken from each equivalence class.

There are two basic possibilities for input partitioning which are

– equivalence classes derived from the specification – the interpretation of the specification may be either input orientated, for example the values selected are treated in the same way, or output orientated, for example the set of values lead to the same functional result;

– equivalence classes derived from the internal structure of the program – 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, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

Static Analysis and Software Assurance. D. Wagner, Lecture Notes in Computer Science, Volume 2126/2001, Springer, 2001, ISBN 978-3-540-42314-0

C.5.8 Structure-based testing

NOTE This technique/measure is referenced in Table B.2 of IEC 61508-3.

Aim: To apply tests which exercise certain subsets of the program structure.

Description: Based on analysis of the program, a set of input data is chosen so that a large (and often prespecified target) percentage of the program code is exercised. Measures of code coverage will vary as follows, depending upon the level of rigour required. In all cases, 100 % of the selected coverage metric should be the aim; if it is not possible to achieve 100 % coverage, the reasons why 100 % cannot be achieved should be documented in the test report (for example, defensive code which can only be entered if a hardware problem arises). The first four techniques in the following list are mentioned specifically in the recommendations in Table B.3 of IEC 61508-3 and are widely supported by testing tools; the remaining techniques could also be considered.

– **Entry point (call graph) coverage**: ensure that every subprogram (subroutine or function) has been called at least once (this is the least rigorous structural coverage measurement).

  NOTE In object-oriented languages, there can be several subprograms of the same name which apply to different variants of a polymorphic type (overriding subprograms) which can be invoked by dynamic dispatching. In these cases every such overriding subprogram should be tested.

– **Statements**: ensure that all statements in the code have been executed at least once.

– **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. See MCDC (modified condition decision coverage, ref. DO178B).

– **LCSAJ**: a linear code sequence and jump is any linear sequence of code statements, including conditional statements, terminated by a jump. Many potential subpaths 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 read.

– **Basis path**: one of a minimal set of finite paths from start to finish, such that all arcs are included. (Overlapping combinations of paths in this basis set can form any path through that part of the program.) Tests of all basis path has been shown to be efficient for locating errors.

References:

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

RTCA, Inc. document DO-178B and EUROCAE document ED-12B, *Software Considerations in Airborne Systems and Equipment Certification*, dated December 1, 1992

C.5.9 Control flow analysis

NOTE This technique/measure is referenced in Table B.8 of IEC 61508-3.

Aim: To detect poor and potentially incorrect program structures.

Description: Control flow analysis is a static testing technique for finding suspect areas of code that do not follow good programming practice. The program is analysed producing a directed graph which can be further analysed for

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

– knotted code. Well-structured code has a control graph which is reducible by successive graph reductions to a single node. In contrast, poorly structured code can only be reduced to a knot composed of several nodes.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

C.5.10 Data flow analysis

NOTE This technique/measure is referenced in Table B.8 of IEC 61508-3.

Aim: To detect poor and potentially incorrect program structures.

Description: Data flow analysis is a static testing technique that 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 may be read before they are assigned a value – this can be avoided by always assigning a value when declaring a new variable;

– 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.

A data flow anomaly will not always directly correspond to a program fault, but if anomalies are avoided the code is less likely to contain faults.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

C.5.11 Symbolic execution

NOTE This technique/measure is referenced in Table B.8 of IEC 61508-3.

Aim: To show the agreement between the source code and the specification.

Description: The program variables are evaluated after 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 expression is a formula for the value that the program would calculate if given real data. This can be checked against the expected expression.

A related use of symbolic execution is as a systematic way of generating test data for the paths through the program logic. The symbolic execution facility may be incorporated into an integrated toolset to provide a facility for software element test and code analysis.

References:

Using symbolic execution for verifying safety-critical systems. A. Coen-Porisini, G. Denaro, C. Ghezzi, M. Pezzé. Proceedings of the 8th European software engineering conference, and 9th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2001, ISBN:1-58113-390-1

Using symbolic execution to guide test generation. G. Lee, J. Morris, K. Parker, G. Bundell, P. Lam. In Software Testing, Verification and Reliability, vol 15, no 1, 2005. John Wiley & Sons, Ltd

C.5.12 Formal proof (verification)

NOTE This technique/measure is referenced in Tables A.5 and A.9 of IEC 61508-3.

Aim: To prove the correctness of a program with respect to some abstract model of the program, using theoretical and mathematical models and rules.

Description: Testing is a common way to examine the correctness of a program. However, exhaustive testing is generally unachievable given the complexity of programs of practical value, and therefore only a fraction of the possible program behaviour can be examined in this way. In contrast, formal verification applies mathematical operations to a mathematical representation of a program in order to establish that the program behaves as defined for all possible inputs.

Formal verification of a system requires an abstract model of the program and of its required behaviour (i.e. a specification) in a language with a precise mathematical meaning. The specification may be complete, or it may be restricted to specific program properties:

- functional correctness properties, i.e. the program should exhibit a particular functionality.
- safety (i.e. some bad behaviour will never occur) and liveness (i.e. some good behaviour will occur eventually) properties.
- timing properties, i.e. some behaviour will occur at a particular time.

The outcome of formal verification is a rigorous argument that the abstract model of the program is correct with respect to the specification for all possible inputs i.e. the model satisfies the specified properties.

However, the correctness of the model does not directly prove the correctness of the actual program, and a further necessary step is to show that the model is an accurate abstraction of the actual program for the properties of interest. Some program properties of practical interest cannot be formalised mathematically (e.g. most timing and scheduling, or subjective properties such as a “clear and simple” user interface, or indeed any property or design

objective that cannot readily be expressed in a formal language). Formal verification therefore does not completely replace simulation and testing, but instead complements these techniques by providing further evidence to support the program's correct operation for all inputs. While formal verification can ensure the correctness of an abstract model of a program, testing ensures that the actual program behaves as expected.

The use of formal verification at the design phase may significantly reduce development time by discovering significant errors and oversights early in the design phase, and thus reducing the time required iterating between design and testing.

Several formal methods in practical use are described in C.2.4: for instance, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z.

C.5.12.1 Model checking

Model checking is a method for the formal verification of reactive and concurrent systems. Given a finite state structure which describes the behaviors of the system, a property written as a temporal logic formula is checked if it holds or not against the structure. Efficient algorithms (e.g. SPIN, SMV, and UPPAAL) are employed to traverse the whole states of the structure automatically and exhaustively. When the property does not hold, a counterexample is generated. It shows how the property is violated in the structure, and contains very useful information to investigate the system. Model checking can detect "deep bugs" that could escape from the traditional inspection and testing.

Note that model checking is helpful in analysing subtle complexity. This may be useful in some low-SIL applications but caution is needed if subtle complexity exists in high-SIL applications.

References:

Is Proof More Cost-Effective Than Testing?. S. King, R. Chapman, J. Hammond, A. Pryor. IEEE Transactions on Software Engineering, vol. 26 no. 8, August 2000

Model Checking. E. M. Clarke, O. Grumberg, and D. A. Peled. MIT Press, 1999, ISBN 0262032708, 9780262032704

Systems and Software Verification: Model-Checking Techniques and Tools. B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. Mckenzie, Springer, 2001, ISBN 3-540-41523-8

Logic in Computer Science: Modelling and Reasoning about Systems. M.Huth and M. Ryan. Cambridge University Press, 2000, ISBN 0521652006, 0521656028

The Spin Model Checker: Primer and Reference Manual. G. J. Holzmann. Addison-Wesley, 2003, ISBN 0321228626, 9780321228628

C.5.12.2 (void)

C.5.13 Complexity metrics

NOTE This technique/measure is referenced in Tables B.9 and C.19 of IEC 61508-3.

Aim: To predict the attributes of programs from properties of the software itself or 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 software module (accessibility) – the more a software module can be accessed, the more likely it is to be debugged;
- Halstead type metrics science – this measure computes the program length by counting the number of operators and operands; it provides a measure of complexity and size that forms a baseline for comparison when estimating future development resources;
- number of entries and exits per software module – minimising the number of entry/exit points is a key feature of structured design and programming techniques.

Reference:

Metrics and Models in Software Quality Engineering. S.H. Kan. Addison-Wesley, 2003, ISBN 0201729156, 9780201729153

C.5.14 Formal inspections

NOTE This technique/measure is referenced in Table B.8 of IEC 61508-3.

Aim: To reveal defects in a software element.

Description: Formal inspection is a structured process to inspect software material that is carried out by peers of the person producing the material to find defects and to enable the producer to improve the material. The producer should take no part in the inspection process, other than to brief the inspectors during the familiarization stage. Formal inspections may be carried out on specific software elements produced at any phase of the software development life-cycle.

Prior to the inspection taking place the inspectors should become familiar with the materials to be inspected. The inspectors' roles in the inspection process should be clearly defined. An inspection agenda should be prepared. Entry and exit criteria should be defined based on the properties required for the software element. Entry criteria are the criteria or requirements which must be met prior to the inspection taking place. Exit criteria are the criteria or requirements which must be met to complete a specific process.

During the inspection the findings of the inspection should be formally recorded by the moderator, whose role is to facilitate the inspection. A consensus on the findings should be reached by all inspectors. Defects should be classified as either a) requiring rectification prior to acceptance or b) requiring rectification by a given time / milestone. Defects identified should be referred to the producer for subsequent rectification after completion of the inspection. Dependent on the number and scope of identified defects, the moderator may determine it to be necessary for a further inspection of the software material.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

Fagan, M. *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal 15, 3 (1976): 182-211

C.5.15 Walk-through (software)

NOTE This technique/measure is referenced in Table B.8 of IEC 61508-3.

Aim: To reveal discrepancies between the specification and implementation.

Description: Walk-through is an informal technique, carried out by the producer of a software element in the presence of his peers with the objective of finding defects in the software element. They may be carried out on specific software elements produced at any phase of the software development life-cycle.

Specified functions of the safety-related system are examined and evaluated to ensure that the safety-related system conforms to the requirements given in the specification. Any points of doubt concerning the implementation and use of the product are documented so they may be resolved. In contrast with a formal inspection, the author is active during the walkthrough procedure.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

C.5.16 Design review

NOTE This technique/measure is referenced in Table B.8 of IEC 61508-3.

Aim: To reveal defects in the design of the software.

Description: A design review is a formal, documented, comprehensive and systematic examination of the software design to evaluate the design requirements and the capability of the design to meet these requirements and identify problems and propose solutions.

Design Reviews provide the means to assess the status of the design against the input requirements, and the means to identify opportunities for further improvement. As the development life-cycle activities progress, and major detailed design milestones are met, Design Reviews should be held to review all interface aspects, ensure that the design can be verified to ensure that the design meets its requirements, and ensure that the most appropriate design is consistent with the safety requirements. Such a review is primarily intended to verify the work of the designers and should be treated as a confirmation and refining activity.

A rigorous inspection technique such as “sneak circuit analysis” may be used to detect incorrect software behaviour such as an unexpected path or logic flow, unintended outputs, incorrect timing, undesired actions.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

The Art of Software Testing, second edition. G. J. Myers, T. Badgett, T. M. Codd, C. Sandler, John Wiley and Sons, 2004, ISBN 0471469122, 9780471469124

IEC 61160:2005, *Design review*

Space Product Assurance, Sneak analysis - Part 2: Clue list. ECSS-Q-40-04A Part 2. ESA Publications Division, Noordwijk, 1997, ISSN 1028-396X, http://www.everyspec.com/ESA/ECSS-Q-40-04A_Part-2_14981/

C.5.17 Prototyping/animation

NOTE This technique/measure is referenced in Tables B.3 and B.5 of IEC 61508-3.

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 subset 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.

Reference:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

C.5.18 Process simulation

NOTE This technique/measure is referenced in Tables A.7, C.7, B.3 and C.13 of IEC 61508-3.

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 equipment under control (EUC).

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

- provide inputs, equivalent to the inputs which will exist when the EUC is actually installed;
- respond to outputs from the software being tested 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:

EmStar: An Environment for Developing Wireless Embedded Systems Software. J Elson et al. http://cens.ucla.edu/TechReports/9_emstar.pdf

A hardware-software co-simulator for embedded system design and debugging. A. Ghosh et al. In Proceedings of the IFIP International Conference on Computer Hardware Description Languages and Their Applications, IFIP International Conference on Very Large Scale Integration, 1995. IEEE, 1995, ISBN 4930813670, 9784930813671

C.5.19 Performance requirements

NOTE This technique/measure is referenced in Table B.6 of IEC 61508-3.

Aim: To establish demonstrable performance requirements of a software system.

Description: An analysis is performed of both the system and the software requirements specifications to specify 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:

- each performance requirement is associated with at least one measurement;
- where possible, accurate and efficient measurements are selected which can be used as early in the development as possible;
- essential and optional performance requirements and success criteria are specified; and
- where possible, advantage should be taken of the possibility of using a single measurement for more than one performance requirement.

Reference:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

C.5.20 Performance modelling

NOTE This technique/measure is referenced in Tables B.2 and B.5 of IEC 61508-3.

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

- producing a model of the system processes, and their interactions;
- determining the use of resources by each process, for example, processor time, communications bandwidth, storage devices, etc;
- determining 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 sufficient, while for more complex systems some form of simulation may be more appropriate 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 (for example 50 %) of the total resources so that the probability of resource starvation is reduced.

Reference:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

C.5.21 Avalanche/stress testing

NOTE This technique/measure is referenced in Table B.6 of IEC 61508-3.

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:

– 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.

Reference:

Software Engineering for Real-time Systems. J. E. Cooling, Pearson Education, 2003, ISBN 0201596202, 9780201596205

C.5.22 Response timing and memory constraints

NOTE This technique/measure is referenced in Table B.6 of IEC 61508-3.

Aim: To ensure that the system will meet 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 to determine 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 benchmarking of time critical systems.

C.5.23 Impact analysis

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-3.

Aim: To determine the effect that a change or an enhancement to a software system will have to other software 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 should be undertaken to determine the impact of the modification or enhancement on the software, and to also determine which software systems and software modules are affected.

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

– only the changed software module is reverified;

– all affected software modules are reverified; or

– the complete system is reverified.

Reference:

Requirements Engineering. E. Hull, K. Jackson, J. Dick. Springer, 2005, ISBN 1852338792, 9781852338794

C.5.24 Software configuration management

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-3.

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 (see IEC 61508-3, 6.2.3). In essence, it requires documenting the production of every version of every significant deliverable and of every relationship between different versions of the different deliverables. The resulting documentation allows the developer to determine the effect on other deliverables of a change to one deliverable (especially one of its elements). In particular, systems or subsystems can be reliably re-built from consistent sets of element versions.

References:

Software engineering: Update. Ian Sommerville, Addison-Wesley Longman, Amsterdam; 8th ed., 2006, ISBN 0321313798, 9780321313799

Software Engineering. Ian Sommerville, Pearson Studium, 8. Auflage, 2007, ISBN 3827372577, 9783827372574

Software Configuration Management: Coordination for Team Productivity. W.A. Babich. Addison-Wesley, 1986, ISBN 0201101610, 9780201101614

CMMI: guidelines for process integration and product improvement, Mary Beth Chrissis, Mike Konrad, Sandy Shrum, Addison-Wesley, 2003, ISBN 0321154967, 9780321154965

C.5.25 Regression validation

NOTE This technique/measure is referenced in Table A.8 of IEC 61508-3.

Aim: To ensure that valid conclusions are drawn from regression testing.

Description: Complete regression testing of a large or complex system will usually require much effort and resource. Where possible, it is desirable to restrict the regression testing to cover only the system aspects of direct interest at that point in the system development. In this partial regression testing it is essential to have a clear understanding of the scope of the partial testing and to draw only valid conclusions regarding the tested state of the system.

Reference:

Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing. R.Black, John Wiley and Sons, 2002, ISBN 0471223980, 9780471223986

C.5.26 Animation of specification and design

NOTE This technique/measure is referenced in Table A.9 of IEC 61508-3.

Aim: To guide the software verification by means of a systematic examination of the specification

Description: A representation of the software that is more abstract than the executable code (i.e. a specification or a high level design) is examined to determine the behaviour of the eventual executable software. The examination is automated in some way (depending on the possibilities afforded by the nature and level of abstraction of the higher level representation) so as to simulate the behaviour and outputs of the executable software. One application of this approach is to generate tests (or “oracles”) that can be later applied to the executable software, thus automating to some degree the testing process. Another application is to animate a user interface so that non-technical end-users can gain some appreciation of the detailed meaning of the specification to which the software developers will work. This provides a valuable method of communication between the two groups.

References:

Supporting the Software Testing Process through Specification Animation. T.Miller, P.Strooper. In Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03), ed. P.Lindsay. IEEE Computer Society, IEEE Computer Society, 2003, ISBN 0769519490, 9780769519494

B model animation for external verification. H.Waeselynck, S.Behnia, In Proceedings. of the Second International Conference on Formal Engineering Methods, 1998. IEEE Computer Society, 1998, ISBN 0-8186-9198-0

C.5.27 Model based testing (test case generation)

NOTE This technique/measure is referenced in Table A.5 of IEC 61508-3.

Aim: To facilitate efficient automatic test case generation from system models and to generate highly repeatable test suites.

Description: Model-based Testing (MBT) is a black-box approach in which common testing tasks such as test case generation (TCG) and test results evaluation are based on a model of the system (application) under test (SUT). Typically, but not only, the systems data and user behaviour are modelled using Finite state machines, Markov processes, decision tables or the like (El-Far, 2001, generalized). Additionally, model-based testing can be combined with source code level test coverage measurement, and functional models can be based on existing source code.

Model-based testing is the automatic generation of efficient test cases/procedures using models of system requirements and specified functionality (SoftwareTech, 2009).

Since testing is very expensive, there is a huge demand for automatic test case generation tools. Therefore, model-based testing is currently a very active field of research, resulting in a

large number of available Test Case Generation (TCG) tools. These tools typically extract a test suite from the behavioural part of the model, guaranteeing to meet certain coverage requirements.

The model is an abstract, partial representation of the desired behaviour of the system under test (SUT). From this model, test models are derived, building an abstract test suite. Test cases are derived from this abstract test suite and executed against the system, and tests can be run against the system model as well. MBT with TCG is based on and strongly related to use of formal methods, so recommendations are similar with respect to safety integrity levels (SIL): HR (highly recommended) for higher SILs, and not required for lower SILs.

The specific activities in general are:

– build the model (from system requirements)
– generate expected inputs
– generate expected outputs
– run tests
– compare actual outputs with expected outputs
– decide on further action (modify model, generate more tests, estimate reliability/quality of the software)

Tests can be derived with different methods and techniques for expressing models of user/system behaviour, e.g.

– by using decision tables
– by using finite state machines
– by using grammars
– by using Markov Chain models
– by using state charts
– by theorem proving
– by constraint logic programming
– by model checking
– by symbolic execution
– by using an event-flow model
– reactive system tests: parallel hierarchical finite automaton
– etc.

Model-based testing is specifically targeting recently the safety critical domain. It allows for early exposure of ambiguities in specification and design, provides the capability to automatically generate many non-repetitive efficient tests, to evaluate regression test suites and to assess software reliability and quality, and eases updating of test suites.

A thorough overview is provided by ElFar (2001) and SoftwareTech 2009 (see references), other details and domain specific issues are discussed in the other references.

References:

T. Bauer, F. Böhr, D. Landmann, T. Beletski, R. Eschbach, Robert and J.H. Poore, *From Requirements to Statistical Testing of Embedded Systems* Software Engineering for Automotive Systems - SEAS 2007, ICSE Workshops, Minneapolis, USA

Eckard Bringmann, Andreas Krämer; *Model-based Testing of Automotive Systems* In: ICST, pp.485-493, 2008 International Conference on Software Testing, Verification, and Validation, 2008

Broy M., *Challenges in automotive software engineering*, International conference on Software engineering (ICSE '06),Shanghai, China, 2006

I. K. El-Far and J. A. Whittaker, *Model-Based Software Testing*. Encyclopedia of Software Engineering (edited by J. J. Marciniak). Wiley, 2001

Heimdahl, M.P.E.: *Model-based testing: challenges ahead*, Computer Software and Applications Conference (COMPSAC 2005), 25-28 July 2005, Edinburgh, Scotland, UK, 2005

Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte, *Model-Based Software Testing and Analysis with C#*, ISBN 978-0-521-68761-4, Cambridge University Press 2008

A. Paradkar, *Case Studies on Fault Detection Effectiveness of Model-based Test Generation Techniques*, in ACM SIGSOFT SW Engineering Notes, Proc. of the first int. workshop on Advances in model-based testing A-MOST '05, Vol. 30 Issue 4. ACM Press 2005

S. J. Prowell, *Using Markov Chain Usage Models to Test Complex Systems*, HICSS '05: 38th Annual Hawaii, International Conference on System Sciences, 2005

Mark Utting and Bruno Legeard, *Practical Model-Based Testing: A Tools Approach*, ISBN 978-0-12-372501-1, Morgan-Kaufmann 2007

Hong Zhu et al. (2008). AST '08: *Proceedings of the 3rd International Workshop on Automation of Software Test*. ACM Press. ISBN 978-1-60558-030-2

Model-Based Testing of Reactive Systems Advanced Lecture Series, LNCS 3472, Springer-Verlag, 2005, ISBN 978-3-540-26278-7

Model-based Testing, SoftwareTech July 2009, Vol. 12, No. 2, Software Testing: A Life Cycle Perspective, http://www.goldpractices.com/practices/mbt/

C.6 Functional safety assessment

NOTE Relevant techniques and measures may also be found in B.6.

C.6.1 Decision tables (truth tables)

NOTE This technique/measure is referenced in Tables A.10 and B.7 of IEC 61508-3.

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

Description: This method uses two dimensional tables to concisely describe logical relationships between Boolean program variables.

The conciseness and tabular nature of the method makes it appropriate as a means of analysing complex logical combinations expressed in code.

The method is potentially executable if used as a specification.

C.6.2 Software Hazard and Operability Study (CHAZOP, FMEA)

Aim: To determine safety hazards in a proposed or existing system, their possible causes and consequences, and recommend action to minimise the chance of their occurrence.

Description: A team of engineers, with expertise covering the whole system under consideration, participate in a structured examination of a design, through a series of scheduled meetings. They consider both the functional aspects of the design and how the system would operate in practice (including human activity and maintenance). A leader encourages team members to be creative in exposing potential hazards, and drives the procedure by presenting each part of the system in connection with several guide words: "none", "more of", "less of", "part of", "more than" (or "as well as") and "other than". Every applied condition or failure mode is considered for its feasibility, how it could arise, the possible consequences (is there a hazard?), how it could be avoided and if the avoidance technique is worth the expense.

Hazard studies may take place at many stages of project development, but are most effective when performed early enough to influence major design and operability decisions.

The HAZOP technique evolved in the process industry and requires modification for software application. Different derivative methods (or Computer HAZOPs – "CHAZOPs") have been proposed which in general introduce new guide words and/or suggest schemes for systematically covering the system and software architecture.

References:

OF-FMEA: an approach to safety analysis of object-oriented software intensive systems, T. Cichocki, J. Gorski. In Artificial Intelligence and Security in Computing Systems: 9th International Conference, ACS '2002. Ed. J. Soldek. Springer, 2003, ISBN 1402073968, 9781402073960

Software FMEA techniques. P.L.Goddard. In Proc Annual 2000 Reliability and Maintainability Symposium, IEEE, 2000, ISBN: 0-7803-5848-1

Software criticality analysis of COTS/SOUP. P.Bishop, T.Clement, S.Guerra. In Reliability Engineering & System Safety, Volume 81, Issue 3, September 2003, Elsevier Ltd., 2003

C.6.3 Common cause failure analysis

NOTE 1 This technique/measure is referenced in Table A.10 of IEC 61508-3.

NOTE 2 See also Annex D of IEC 61508-6.

Aim: To determine potential failures in multiple systems or multiple subsystems which would undermine the benefits of redundancy, because of the appearance of the same failures in the multiple parts at the same time.

Description: Systems intended to take care of the safety of a plant often use redundancy in hardware and majority voting. This is to avoid random hardware failures in components or subsystems which would tend to prevent the correct processing of data.

However, some failures can be common to more than one component or subsystem. For example, if a 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 system such as fire, flooding, electromagnetic interference, plane crashes, and earthquakes. The 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, and operating and maintenance personnel are extensively trained.

Internal effects are also major contributors to common cause failures. They can stem from design faults in common or identical components and their interfaces, as well as ageing of components. Common cause failure analysis has to search the system for such potential common failures. Methods of common cause failure 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 software diversity is used in different channels of a redundant system, there might be some commonality in the software approaches which could give rise to common cause failure, for example, faults in the common specification.

When common cause failures do not occur exactly at the same time, precautions can be taken by means of comparison methods between the multiple channels which should lead to detection of a failure before this failure is common to all channels. Common cause failure analysis should take this technique into account.

References:

Reliability analysis of hierarchical computer-based systems subject to common-cause failures. L.Xing, L.Meshkat, S.Donohue. Reliability Engineering & System Safety Volume 92, Issue 3, March 2007

C.6.4 Reliability block diagrams

NOTE 1 This technique/measure is referenced in Table A.10 of IEC 61508-3 and is used in Annex B of IEC 61508-6.

NOTE 2 See also B.6.6.7 “Reliability block diagrams”.

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:

IEC 61025:2006, *Fault tree analysis (FTA)*

From safety analysis to software requirements. K.M. Hansen, A.P. Ravn, A.P, V Stavridou. IEEE Trans Software Engineering, Volume 24, Issue 7, Jul 1998

IEC 61078:2006, *Analysis techniques for dependability – Reliability block diagram and boolean methods*

Annex D
(informative)

A probabilistic approach to determining software safety integrity for pre-developed software

D.1 General

This annex provides initial guidelines on the use of a probabilistic approach to determining software safety integrity for pre-developed software based on operational experience. This approach is considered particularly appropriate as part of the qualification of operating systems, library modules, compilers and other system software. The annex provides an indication of what is possible, but the techniques should be used only by those who are competent in statistical analysis.

NOTE This annex uses the term confidence level, which is described in IEEE 352. An equivalent term, significance level, is used in IEC 61164.

The techniques could also be used to demonstrate an increase in the safety integrity level of software over time. For example, software built to the requirements of IEC 61508-3 to SIL1 may, after a suitable period of successful operation in a large number of applications, be shown to achieve SIL2.

Table D.1 below shows the number of failure-free demands experienced or hours of failure-free operation needed to qualify for a particular safety integrity level. This table is a summary of the results given in D.2.1 and D.2.3.

Operating experience can be treated mathematically as outlined in D.2 below to supplement or replace statistical testing, and operating experience from several sites may be combined (i.e. by adding the number of treated demands or hours of operation), but only if

– the software version to be used in the E/E/PE safety-related system is identical to the version for which operating experience is being claimed;
– the operational profile of the input space is similar;
– there is an effective system for reporting and documenting failures; and
– the relevant prerequisites (see D.2 below) are satisfied.

Table D.1 – Necessary history for confidence to safety integrity levels

SIL	Low demand mode of operation	Number of treated demands		High demand or continuous mode of operation	Hours of operation in total	
	(Probability of failure to perform its design function on demand)	$1-\alpha = 0,99$	$1-\alpha = 0,95$	(Probability of a dangerous failure per hour)	$1-\alpha = 0,99$	$1-\alpha = 0,95$
4	$\geq 10^{-5}$ to $< 10^{-4}$	$4,6 \times 10^5$	3×10^5	$\geq 10^{-9}$ to $< 10^{-8}$	$4,6 \times 10^9$	3×10^9
3	$\geq 10^{-4}$ to $< 10^{-3}$	$4,6 \times 10^4$	3×10^4	$\geq 10^{-8}$ to $< 10^{-7}$	$4,6 \times 10^8$	3×10^8
2	$\geq 10^{-3}$ to $< 10^{-2}$	$4,6 \times 10^3$	3×10^3	$\geq 10^{-7}$ to $< 10^{-6}$	$4,6 \times 10^7$	3×10^7
1	$\geq 10^{-2}$ to $< 10^{-1}$	$4,6 \times 10^2$	3×10^2	$\geq 10^{-6}$ to $< 10^{-5}$	$4,6 \times 10^6$	3×10^6

NOTE 1 $1-\alpha$ represents the confidence level.

NOTE 2 See D.2.1 and D.2.3 for prerequisites and details of how this table is derived.

D.2 Statistical testing formulae and examples of their use

D.2.1 Simple statistical test for low demand mode of operation

D.2.1.1 Prerequisites

a) Test data distribution equal to distribution for demands during on-line operation.

b) Test runs are statistically independent from each other, with respect to the cause of a failure.

c) An adequate mechanism exists to detect any failures which may occur.

d) Number of test cases $n > 100$.

e) No failure occurs during the n test cases.

D.2.1.2 Results

Failure probability p (per demand), at the confidence level $1-\alpha$, is given by

$$p \leq 1 - \sqrt[n]{\alpha} \qquad \text{or} \qquad n \geq -\frac{\ln \alpha}{p}$$

D.2.1.3 Example

Table D.2 – Probabilities of failure for low demand mode of operation

$1-\alpha$	P
0,95	$3/n$
0,99	$4{,}6/n$

For a probability of failure on demand of SIL 3 at 95 % confidence the application of the formula gives 30 000 test cases under the conditions of the prerequisites. Table D.1 summarises the results for each safety integrity level.

D.2.2 Testing of an input space (domain) for a low demand mode of operation

D.2.2.1 Prerequisites

The only prerequisite is that the test data is selected to give a random uniform distribution over the input space (domain).

D.2.2.2 Results

The objective is to find the number of tests, n, that are necessary based on the threshold of accuracy, δ, of the inputs for the low demand function (such as a safety shut-down) that is being tested.

Table D.3 – Mean distances of two test points

Dimension of the domain	Mean distance of two test points in direction of an arbitrary axis
1	$\delta = 1/n$
2	$\delta = \sqrt[2]{1/n}$
3	$\delta = \sqrt[3]{1/n}$
k	$\delta = \sqrt[k]{1/n}$
NOTE *k* can be any positive integer. The values 1, 2 and 3 are just examples.	

D.2.2.3 Example

Consider a safety shut-down that is dependent on just two variables, A and B. If it has been verified that the thresholds that partition the input pair of variables A and B are treated correctly to an accuracy of 1 % of A or B's measuring range, the number of uniformly distributed test cases required in the space of A and B is

$$n = 1/\delta^2 = 10^4$$

D.2.3 Simple statistical test for high demand or continuous mode of operation

D.2.3.1 Prerequisites

a) Test data distribution equal to distribution during on-line operation.

b) The relative reduction for the probability of no failure is proportional to the length of the considered time interval and constant otherwise.

c) An adequate mechanism exists to detect any failures which may occur.

d) The test extends over a test time *t*.

e) No failure occurs during *t*.

D.2.3.2 Results

The relationship between the probability of failure λ, the confidence level 1-α and the testing time *t* is

$$\lambda = -\frac{\ln \alpha}{t}$$

The probability of failure is indirectly proportional to the mean operating time between failures:

$$\lambda = \frac{1}{\text{MTBF}}$$

NOTE This standard does not distinguish between the probability of failure per hour and the rate of failures in 1 h. Strictly, the probability of failure, *F*, is related to the failure rate, *f*, by the equation $F = 1\text{-}e^{-ft}$, but the scope of this standard is for failure rates of less than 10^{-5}, and for values this small $F \approx ft$.

D.2.3.3 Example

Table D.4 – Probabilities of failure for high demand or continuous mode of operation

$1-\alpha$	λ
0,95	$3/t$
0,99	$4{,}6/t$

To verify that the mean time between failures is at least 10^8 h with a confidence level of 95 %, a test time of 3×10^8 h is required and the prerequisites must be satisfied. Table D.1 summarises the number of tests required for each safety integrity level.

D.2.4 Complete test

The program is considered as an urn containing a known number *N* of balls. Each ball represents a program property of interest. Balls are drawn at random and replaced after inspection. A complete test is achieved if all the balls are drawn.

D.2.4.1 Prerequisites

a) Test data distribution is such that each of the *N* program properties is tested with equal probability.

b) Test runs are independent from each other.

c) Each occurring failure is detected.

d) Number of test cases $n >> N$.

e) No failure occurs during the *n* test cases.

f) Each test run tests one program property (a program property is what can be tested during one run).

D.2.4.2 Results

The probability *p* to test all program properties is given by

$$p = \sum_{j=0}^{N-1} (-1)^j \binom{N}{j} \left(\frac{N-j}{N}\right)^n \quad \text{or} \quad p = 1 + \sum_{j=1}^{N} (-1)^j \, C_{j,N} \left(\frac{N-j}{N}\right)^n$$

where

$$C_{j,N} = \frac{N(N-1)\ldots(N-j+1)}{j!}$$

For evaluation of this formula usually only the first terms matter since realistic cases are characterised by $n >> N$. The last factor makes all terms for large *j* very small. This is also visible in Table D.5.

D.2.4.3 Example

Consider a program that has been used at several installations for several years. In total, at least $7{,}5 \times 10^6$ runs have been executed. It is estimated that each 100th run fulfils the above prerequisites. So $7{,}5 \times 10^4$ runs made can be taken for statistical evaluation. It is estimated that 4 000 test runs would perform an exhaustive test. The estimates are conservative. According to Table D.5, the probability of not having tested everything equals $2{,}87 \times 10^{-5}$.

For N = 4 000, the values of the first terms depending on *n* are:

Table D.5 – Probability of testing all program properties

n	*P*
5×10^4	$1 - 1{,}49 \times 10^{-2} + 1{,}10 \times 10^{-4} - \ldots$
$7{,}5 \times 10^4$	$1 - 2{,}87 \times 10^{-5} + 4 \times 10^{-10} - \ldots$
1×10^5	$1 - 5{,}54 \times 10^{-8} + 1{,}52 \times 10^{-15} - \ldots$
2×10^5	$1 - 7{,}67 \times 10^{-19} + 2{,}9 \times 10^{-37} - \ldots$

In practice, such estimates should be made so that they are conservative.

D.3 References

Further information on the above techniques can be found in the following documents:

IEC 61164:2004, *Reliability growth – Statistical test and estimation methods*

Verification and Validation of Real-Time Software, Chapter 5. W. J. Quirk (ed.). Springer Verlag, 1985, ISBN 3-540-15102-8

Combining Probabilistic and Deterministic Verification Efforts. W. D. Ehrenberger, SAFECOMP 92, Pergamon Press, ISBN 0-08-041893-7

Ingenieurstatistik. Heinhold/Gaede, Oldenburg, 1972, ISBN 3-486-31743-1

IEEE 352:1987, *IEEE Guide for general principles of reliability analysis of nuclear power generating station safety systems*

Annex E
(informative)

Overview of techniques and measures for design of ASICs

NOTE The overview of techniques and measures contained in this annex and referenced by IEC 61508-2. This annex should not be regarded as either complete or exhaustive.

E.1 Design description in (V)HDL

Aim: Functional description at high level in hardware description language, for example VHDL or Verilog.

Description: Functional description at high abstraction level in hardware description language, for example VHDL or Verilog. The applied hardware description language should allow functional and/or application oriented description and should be abstracted from later implementation details. Dataflows, branches, arithmetical and/or logical operations should be implemented by assignment and operators of the hardware description language, without manual conversion in logical gates of the applied library.

NOTE For simplification “functional description at high abstraction level in hardware description language” will be denoted in the rest of the document as (V)HDL.

Reference:

IEEE VHDL, *Verilog + Standard VHDL Design guide*

E.2 Schematic entry

Aim: Functional description of the circuitry by drawing a circuit plan using gates and/or macros of the vendor library.

Description: Description of the circuit functionality by schematic entry of the circuit plan. The function to be realised should be implemented by instancing (import) the elementary logical circuit elements such as AND, OR, NOT along with macros consisting of complex arithmetical and logical functions, which are then interconnected. Complex circuits should be partitioned considering the functional viewpoints and can be distributed on different drawings, which are hierarchically interconnected. The interconnection signals should be uniquely defined and have explicit signal names over the entire hierarchy. The use of global signals (Labels) should be avoided as far as applicable.

E.3 Structured description

NOTE See also C.2.7 "Structured Programming" and E.12 "Modularization".

Aim: The description of the circuit's functionality should be structured in such a fashion that it is easily readable, i.e. circuit function can be intuitively understood on basis of description without simulation efforts.

Description: Description of the circuit functionality with (V)HDL or by schematic entry. An easily recognisable and modular structure is recommended. Each module should be implemented likewise in the same fashion and should be described in such a way that it is easily readable with clear defined sub functions. A strict distinction between implemented function and interconnection is recommended, i.e. the module, which is implemented by instancing other sub modules, contains explicitly interconnections of the instanced modules and should not contain any circuit logic.

E.4 Proven-in-use tools

Aim: Application of proven-in-use tools to avoid systematic failure by sufficient long-approved practice of the tools in various projects.

Description: Most of the used tools for designing ASICs and FPGAs comprise of sophisticated software, which cannot be considered to operate without any faults with respect to its correct functionality and it is also quite likely that faults might occur due to faulty operation. Therefore only tools with the attribute "proven-in-use" should be preferred for designing ASICs and FPGAs. This implies:

– Application of tools which have been used (in a comparable software version) over a long period of time or high number of users in various projects with equivalent complexity.
– Adequate experience of each ASIC/FPGA designer with the operation of the tool over a long period of time.
– Use of commonly used tools (adequate number of users) so that information regarding known failures with work arounds (version control with "Bug-List") is available. This information should be readily integrated in design flow and helps to avoid systematic failures.
– The consistency check of the internal tool database and the plausibility check avoid faulty output data. Standard tools check the consistency of the internal database, for example the consistency of database between synthesis- and place-and-route-tool, in order to operate with unique data.

NOTE The consistency check is an inherent attribute of the tool under use and the designer has limited influence on it. Therefore, if the possibility of manual consistency check is provided, the designer should use it adequately.

E.5 (V)HDL simulation

NOTE See also E.6 "Functional test on module level".

Aim: Functional verification of circuit described in (V)HDL by means of simulation.

Description: Verification of the function by simulating the entire circuit or each sub module. The (V)HDL simulator detects a sequence of outputs caused by the internal change of the circuit states as the result of applied input stimuli. The verification of the detected output sequence can be carried out either by pretraced sequence of output signals ("Wave form") or by a special environment known as test bench, which is installed during the design process. The chosen simulator should have an attribute "proven-in-use" in order to provide correct results and to mask faulty timing behaviour of the signals (Spikes, tri-state tracing), which might be caused by the simulator itself or faulty modelling.

E.6 Functional test on module level

NOTE See also E.5 "(V)HDL Simulation" and E.13 "Coverage of the verification scenarios".

Aim: Functional verification "Bottom-up".

Description: Verification of the implemented function - for example by simulation - at module level. The module under test will be instanced in a typical virtual test environment known as ”test bench” and stimulated by the test pattern contained in the code. A sufficient high coverage of specified function including all special cases if they exist is at least required. Automatic verification of output sequence by the code of "test bench" should be preferred against manual inspection of output signals.

E.7 Functional test on top level

NOTE See also E.8 "Functional test embedded in system environment”.

Aim: Verification of the ASIC (entire circuit).

Description: The objective of the test is the verification of the entire circuit (ASIC).

E.8 Functional test embedded in system environment

NOTE See also E.7 "Functional test on top level".

Aim: Verification of the specified function embedded in system environment.

Description: This test will verify the entire functionality of the circuit (ASIC) in its system environment, for example with all other components that are located on the circuit boards or elsewhere. A modelling of all relevant components on the circuit board and simulation of ASIC together with the created model to verify the correct functionality inclusive of timing behaviour is recommended. A complete functional test includes also testing of modules that are activated only during presence of failure.

E.9 Restricted use of asynchronous constructs

Aim: Avoidance of typical timing problems during synthesis, avoidance of ambiguity during simulation and synthesis caused by insufficient modelling, design for testability.

Description: Asynchronous constructs such as SET and RESET signals derived over combinatorial logic are susceptible during synthesis and produce circuits with spikes or inverse timing sequence and therefore should be avoided. Also insufficient modelling may not be interpreted properly by the synthesis tool, which causes ambiguous results during simulation. Additionally asynchronous constructs are poorly testable or not at all testable, so that the test coverage of production and on-line test is effectively reduced. The implementation of completely synchronous design with limited number of clock signals is therefore recommended. In systems with multiphase clocks, all the clocks should be derived from one central clock. Clock input of sequential logic should be always supplied exclusively by the clock signal, which does not contain any combinatorial logic. Asynchronous SET and RESET inputs of sequential cells should be always supplied by synchronous signals that do not contain any combinatorial logic. Master SET and RESET should be synchronised using two Flip-flops.

E.10 Synchronisation of primary inputs and control of metastabilities

Aim: Avoidance of ambiguous circuit behaviour as a result of set-up and hold timing violation.

Description: Input signals from external peripherals are generally asynchronous and can change their state arbitrarily. A direct processing of such signals by the synchronous sequential circuit elements of ASIC/FPGA, for example flip-flops leads to set-up and hold time violation resulting in unpredictable timing and functional behaviour of the ASIC/FPGA. Ultimately the metastability of the memory element might occur. Each asynchronous input signal should be therefore synchronised with respect to the synchronous ASIC circuit to avoid the functional ambiguity. Following measures are recommended:

– Input signals should be synchronised with two consecutive memory elements (Flip-flops) or some equivalent circuit in order to achieve a predictable functional behaviour.

– Each asynchronous input signal should be fundamentally synchronised in the above defined manner, i.e. each asynchronous signal is connected with exact one such synchronising circuit. If necessary the output of the synchronising circuit can be used for multiple access.

– The synchronising circuit should be used for stability test of parallel bus signals and to control the data consistency near sampling point

E.11 Design for testability

NOTE 1 See also E.31 "Implementation of test structures".

Aim: Avoidance of not testable or poorly testable structures in order to achieve high test coverage for production test or on-line test.

Description: Design for testability is governed by the avoidance of

– asynchronous constructs
– latches and on-chip tri-state signals
– wired-and / wired-or logic and redundant logic.

The combinatorial depth of the sub circuits plays an important role during the testing. The test pattern required for a complete test increases exponentially with the combinatorial depth of the circuit. Therefore, circuits with high combinatorial depth are only poorly testable with adequate means.

A design for testability orientated approach ensures that the desired test coverage is achieved. As the actual test coverage can be determined at a very late stage in the design process, insufficient consideration of “design for testability” issues might dramatically reduce the achievable test coverage, leading to additional effort.

NOTE 2 The test coverage is usually determined by the percentage of stuck-at faults detected.

E.12 Modularisation

NOTE See also C.2.8 "Information hiding/encapsulation", C.2.9 "Modular approach" and E.3 "Structured description".

Aim: Modular description of the circuit functions.

Description: Distinct partitioning of the total functionality in different modules with limited functions. So the transparency of the modules with the precisely defined interface is established. Every subsystem, at all levels of the design, is clearly defined and is of restricted size (only a few functions). The interfaces between subsystems are kept as simple as possible and the cross-section (i.e. shared data, exchange of information) is minimised. The complexity of individual subsystems is also restricted.

E.13 Coverage of the verification scenarios (test benches)

Aim: Quantitative and qualitative assessment of the applied verification scenarios during the functional test.

Description: The quality of the verification scenarios that is defined during the functional test, i.e. the applied test pattern (stimuli) to verify specified function including all special cases, if they exist, should be qualitatively and/or quantitatively documented. During a quantitative approach the achieved test coverage and the depth of the applied functional tests should be documented. The resulting coverage should meet the levels established for each of the coverage metrics. Any exception will be documented. In the case of a qualitative approach, the number of verified code lines, instructions or paths (“Code coverage”) of the circuit code to be verified should be estimated.

NOTE Exclusive ”Code coverage”-analysis has only a limited relevance, because of high parallelism of the hardware description, and will be justified by exhaustive checks. The code coverage” generally serves to demonstrate the not covered functional code.

E.14 Observation of coding guidelines

Aim: Strict observation of the coding style results in a syntactic and semantic correct circuit code.

Description: Syntactic coding rules help to create an easily readable code and allow a better documentation including version control. Typically, the rules for organising and commenting the instruction blocks or modules can be mentioned here.

Semantic coding rules help avoiding typical implementation problems by avoidance of constructs that lead to faulty synthesis with ambiguous implementation of the circuit function. Typical rules are for example the avoidance of asynchronous constructs or constructs that produce unpredictable timing sequence. In general the use of latches or coupling of data with clock signals lead to such ambiguities.

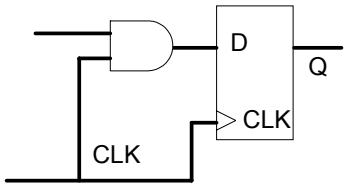
Design guidelines are recommended to avoid systematic design failures during ASIC development process. A coding style in certain aspect limits the design efficiency, offers however in turn the advantage of failure avoidance during ASIC development process. These are in particular:

– avoidance of typical coding infirmity or failure;
– restrictive usage of problematic constructs that produce ambiguous synthesis results;
– design for testability;
– transparent and easy to use code.

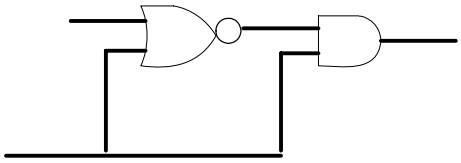
Example of a Coding Style

1. The code should contain as many comments as necessary to understand the function and implementation details. The used conventions have to be defined before the beginning of the design. The compliance of defined conventions should be checked during the design phase.
   1.1. Standard headers include history, cross references to specification, responsibility and design accompanying data such as version number, change requests etc.
   1.2. Easily readable templates: equivalent processes should be described with the same procedure, i.e. usage of predefined templates for recurrent processes (if-then-else, for etc.).
   1.3. Precise and readable naming convention e.g. capital/small letter, pre- and postfix, precise differentiation between port name, internal signals, constants, variables, low active level (xxx_n) etc.
   1.4. Module size restriction and number of ports per module should be limited to increase the readability of code.
   1.5. Structural and defensive code development, e.g. state information should be encapsulated in FSM (Information hiding) to provide the easy alteration of code.
   1.6. Plausibility checks such as range checking etc. should be implemented.
   1.7. Avoidance of following constructs/instructions
      – use of ascending range (x to y) for bus signals;
      – “Disable” instruction in Verilog (corresponds the instruction goto);
      – multidimensional arrays (> 2), records;
      – combination of signed and unsigned data type.
2. Complete synchronous design (clocks derived from central clocks are allowed)
   2.1. Module outputs should be synchronised, it also supports the testability and static timing analysis.
   2.2. Gated clocks should be handled with special precaution.

3. Avoidance of the coupling of data with clock increases the testability, reproducibility between pre- and post-layout data and compliance with RTL (register transfer level) behaviour.
4. Redundant logic is not testable and should be avoided:



Coupling of data and clock signals



Redundant logic

5. Feedback loops in the combinatorial logic should be avoided because they produce unstable design and will not be testable.
6. Full-scan design is recommended.
7. Avoidance of latches increases the testability and reduces the timing constraints during synthesis.
8. Master reset and all asynchronous inputs should be synchronised with two consecutive memory elements (Flip-flops) or an equivalent circuit(s) (metastability).
9. It is recommended to avoid asynchronous set/reset except for master reset.
10. The signals at the module port level should be of the type std_logic or std_logic_vector.

E.15 Application of code checker

Aim: Automatic verification of coding rules ("Coding style") by code checker tool.

Description: The application of code checker helps to a great extent automatically to observe the coding style and generates on-line documentation. However automatic code checker can generally verify the syntax and semantic of the code. The application of such tools should be therefore accompanied by the extension of general coding rules ("tool specific") with the project specific coding rules that the designer has to implement and evaluate separately.

E.16 Defensive programming

See C.2.5.

E.17 Documentation of simulation results

Aim: Documentation of all data needed for a successful simulation in order to verify the specified circuit function.

Description: All the data needed for the functional simulation at module-, chip- or system level should be well documented and archived with the following aims:

– To repeat the simulation at any later phase in turn key fashion.
– To demonstrate the correctness and completeness of all functions specified.

The following database should be archived for this purpose:

– Simulation set-up including complete software of the applied tools, for example simulator, synthesiser with corresponding version and the necessary simulation library.
– Log file of the simulation with full details regarding the time of simulation, applied tools with version and complete report of the work around if it was necessary.
– All relevant simulation results inclusive signal flow, especially in case of manual inspection and documentation of acquired results.

E.18 Code-Inspection

NOTE 1 See also C.5.14 "Formal inspections".

Aim: Review of circuit description.

Description: Review of circuit description should be carried out by

– Checking the coding style.
– Verification of the described functionality against the specification.
– Checking for defensive coding, error and exception handling.

NOTE 2 If the (V)HDL Simulation is not carried out, the completeness of the code inspection and the achieved results should have the equivalent quality that would be achieved by a (V)HDL-simulation.

E.19 Walk-through

NOTE 1 See also C.5.15 "Walk-through".

Aim: Review of the circuit description by walk-through.

Description: A code walk-through consists of a walk-through team selecting a small set of 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.

NOTE 2 As stand alone measure it should be applied only to the circuits with very low complexity. In the case of the failing (V)HDL-Simulation the completeness of the walk-through and the quality of the achieved results should have the equivalent quality that will be achieved by a (V)HDL-simulation.

Reference:

IEC 61160:2005, *Design review*

E.20 Application of validated soft cores

Aim: Avoidance of failure during the operation of soft cores by application of validated soft cores.

Description: If the vendor validates the soft core, following requirements should be fulfilled:

– The validation of the soft core should be carried out for the operation of the safety related system, having at least an equivalent or higher safety integrity level than the system under plan.
– All the assumptions and confinements, which are necessary for the validation of the soft core, should be complied.
– All the necessary documents for the validation of the soft core should be easily available, see also E.17 "Documentation of simulation results".
– Each vendor specification should be strictly observed and the evidence of the compliance should be documented.

E.21 Validation of the soft core

NOTE See also E.6 "Functional test on module level".

Aim: Avoidance of failure during the operation of the soft core by validation of the soft core during design life cycle.

Description: If the soft core is not explicitly developed for the operation in a safety related system, the generated code should be validated under the same premises that apply for the validation of any source code. This means that all possible test cases should be defined and implemented. The functional verification should be then derived by simulation.

E.22 Simulation of the gate netlist to check timing constraints

Aim: Independent verification of the achieved timing constraint during synthesis.

Description: Simulation of the gate netlist produced by the synthesis including the back-annotation of line delays and gate delays. The stimuli should be derived to stimulate the circuit in such a fashion that it will cover a high percentage of the timing constraints and include all the worst case timing paths. In general, the stimuli needed to perform E.6 "Functional test on module level" or E.7 "Functional test" provide a suitable criteria for the selection of the stimuli, provided sufficient test coverage can be claimed during the functional test. The circuit should be tested under best- and worst-case condition at the maximum specified clock rate.

The timing verification can be carried out by the automatic check of the set-up and hold time of the memory elements (flip-flops) of the target library as well as by the functional verification of the circuit. The functional verification should be primarily performed by observing the outputs of the chip. This can be automated by comparing the output signals of the circuit with an adequate reference model or (V)HDL source code of the circuit. This test is known as a “regression test” and should be preferred against a manual check of the output signals.

NOTE By applying this measure, the timing behaviour of only those paths can be verified which are actually stimulated during the simulation and, therefore, the bespoke measure cannot provide a complete timing analysis of the circuit in general.

E.23 Static analysis of the propagation delay (STA)

Aim: Independent verification of the timing constraints realised during the synthesis.

Description: Static Timing Analysis (STA) analyses all the paths of a netlist (circuit) generated by the synthesis tool considering the back-annotation, i.e. estimated line delays by the synthesis tool, as well as gate delays without performing the actual simulation. Therefore it allows in general a complete analysis of the timing constraint of the entire circuit. The circuit to be tested should be analysed under best- and worst-case condition operating at maximum specified clock rate and accounting for applicable clock jitter and duty cycle skew. The number of non-relevant timing paths can be limited to a certain minimum by adopting a suitable design technique. It is recommended to investigate, analyse and define the used technique that allows easily readable results before beginning with the design.

NOTE It can be assumed that STA covers explicit all the existing timing paths if

a) The timing constraints are properly specified.

b) The circuit under test contains only such timing paths that can be analysed by STA tools, i.e. generally the case with full synchronous circuits.

E.24 Verification of the gate netlist against reference model by simulation

Aim: Functional equivalence check of the synthesised gate netlist.

Description: Simulation of the gate netlist generated by synthesis tool. The applied stimuli for the verification of the circuit by simulation correspond exactly to the stimuli applied during the E.6 "Functional test on module level" and the E.7 "Functional test on top level" for the verification of the function at module level and top level respectively. The functional verification should be primarily performed by observing the outputs of the chip. This can be automated by comparing the output signals of the circuit with an adequate reference model or (V)HDL source code of the circuit. This test is known as a “regression test” and should be preferred to a manual check of the output signals.

NOTE By applying this measure the functional behaviour of only those paths are verified which are actually stimulated during the simulation. The test coverage can, therefore, only be as good as during the original functional test at module- or top-level, respectively. It is possible to complement this measure with a formal equivalence test. In all cases a functional verification of the (V)HDL source code should be carried out with the final netlist generated by the synthesis tool.

E.25 Comparison of the gate netlist with the reference model (formal equivalence test)

Aim: Functional equivalence check which is independent of simulation.

Description: Comparison of the circuit functionality described by the (V)HDL source code with the circuit functionality of the gate netlist generated by synthesis. The tools based on the formal equivalence principle are capable of verifying the functional equivalence of a different representation form of the circuit for example (V)HDL description or netlist description. By applying this measure a functional simulation is not necessary and an independent functional check is feasible. The successful application of this measure can only be guaranteed, if the applied tool is capable of proving complete equivalence and all the discrepancies reported are evaluated either by manual inspection or automatically.

NOTE It is advantageous to combine this measure with E.24 "Verification of the gate netlist against reference model by simulation". In all cases, a functional verification of (V)HDL source code should be carried out with the final netlist generated by the synthesis tool.

E.26 Check of vendor requirements and constraints

Aim: Avoidance of failure during production by checking the vendor requirements.

Description: A careful checking of the vendor requirements and constraints for example minimum and maximum fan-in and fan-out, maximum wire length (line delay), maximum slew rate of the signals, clock skew and so on by the synthesis tool enhances the reliability of the product. Besides the importance of the requirements for the production process, their violation has a great impact on the validity of the applied models that are used for the simulation. So that any violation of the vendor requirements and constraints leads to faulty simulation results producing undesired functionality.

E.27 Documentation of synthesis constraint, results and tools

Aim: Documentation of all defined constraints that are necessary for an optimal synthesis to generate the final gate netlist.

Description: The documentation of all the synthesis constraints and results is indispensable because of the following reasons:

– to reproduce the synthesis at any later phase.
– to generate an independent synthesis results for verification.

Essential documents are:

– Synthesis set-up including the applied tools and synthesis software with the actual version, the applied synthesis library and the defined constraints and scripts.
– Synthesis log file with the time remark, applied tool with version and complete documentation of the synthesis.
– The generated netlist with estimated time delays (standard delay format (SDF) File).

E.28 Application of proven-in-use synthesis tool

Aim: Tool based conversion of (V)HDL description of a circuit in gate netlist.

Description: Tool based mapping of the (V)HDL source code of circuit functionality by connection of the suitable gates and circuit primitives of the target ASIC library. The selected implementation out of a variety of possible implementations that fulfil the desired functionality depends on the most optimal result that is derived by the synthesis constraints such as timing (clock rate) and chip area.

E.29 Application of proven-in-use target library

NOTE See also E.4 "Proven-in-use tools".

Aim: Avoidance of systematic failures caused by a faulty target library.

Description: The synthesis and simulation target library for the development of an ASIC are derived from a common database and, therefore, are not independent. A systematic failure such as:

– ambiguity between real and modelled behaviour of the circuit elements,
– insufficient modelling for example of set-up and hold time,

is one of the typical examples.

Therefore only “proven-in-use” technologies and target libraries should be used for the design of ASICs that perform safety functions. This means:

– The application of target libraries that have been used over a significant long time in projects with comparable complexity and clock rating.
– Availability of the technology and corresponding target library over a sufficient long period, so that enough modelling accuracy of the library can be expected.

E.30 Script based procedures

Aim: Reproducibility of results and automation of the synthesis cycles.

Description: Automatic and script based control of the synthesis cycles including the definition of the applied constraints. Besides a precise documentation of a complete synthesis constraint, it helps to reproduce the netlist after the alteration of the (V)HDL source code under identical conditions.

E.31 Implementation of test structures

Aim: Design of testable ASICs that guarantees the final production test.

Description: Design for testability allows easily testable circuits by implementation of different test structures, for example:

– Scan-path: In a scan technique, either all (full scan design) or part of flip-flops (partial scan design) is connected in a single chain or multiple chains building a chain of shift registers. The scan-path allows an automatic generation of test pattern of the entire logic of a circuit. The tool generating test pattern is called ATPG “Automatic Test Pattern Generator”. The implementation of scan-path improves the testability of a circuit tremendously and allows more than 98 % of test coverage with reasonable effort. It is therefore recommended to implement, if possible, a full scan-path.

– NAND-Tree: In a NAND-tree, all the primary inputs of a circuit are connected in cascading fashion to build a chain. By applying a suitable test pattern (“walking bit”) it is possible to test the switching behaviour (timing and triggering level) of the inputs. NAND-tree offers a straightforward means for the characterisation of primary inputs. Its implementation is recommended, if the switching behaviour of the circuit cannot be tested otherwise.

– Build-in self test (BIST): Self test of the circuit and in particular the self test of the embedded memory can be carried out very efficiently by implementing on-chip test pattern generator. BIST allows an automatic verification of the circuit structure by applying a pseudo-random test pattern and evaluating the signature of the implemented circuit structure. BIST is recommended as additive measure particularly for memory test. The scan-path test can be replaced by a BIST.

– Quiescent current test (IDDQ-test): A static CMOS-circuit consumes a current mainly during switching event. An absolutely defect free circuit consumes therefore negligibly small amount of current (< 1μA, leakage current) as long as the test pattern is held stationary. IDDQ-test is very effective and provides more than 50 % test coverage just after the application of a couple of test patterns. IDDQ-test can be applied on functional test patterns as well as on synthesised test patterns generated by ATPG. This test method has proven to be most helpful in practice and is capable of detecting failure that other tests rarely or even cannot detect. This measure should therefore be applied additive to the regular production tests.

– Boundary-scan: Test architecture implemented for the verification of the interconnection of the components on a printed circuit board according to JTAG standard. Same philosophy can also be applied to verify the interconnection of modules on chip level. Boundary-scan is primarily recommended to improve the testability of the printed circuit board.

E.32 Estimation of test coverage by simulation

Aim: Determination of the achieved test coverage by the implemented test architecture during production test.

Description: Test coverage achieved by the scan-path test, BIST, functional test pattern or any other measures can be determined by fault simulation. During the fault simulation a test pattern is applied to a circuit in which the faults are inserted. A faulty response of the circuit to the applied stimuli corresponds to the faults inserted and thus contributes to the test coverage. The fault simulation allows the detection of stuck-at-faults ”stuck-at-1” and ”stuck-at-0” and the achieved test coverage represents the quality of the test pattern applied. The fault simulation in general can be used very effectively to detect faults associated with logic that is not a part of the scan-path, for example in case of partial scan-paths.

E.33 Estimation of the test coverage by application of ATPG tool

Aim: Determination of the test coverage that can be expected by synthesised test pattern (Scan-path, BIST) during the production test.

Description: Currently, there is variety of procedures, which generate pseudo-random or algorithmic test patterns for a circuit implemented with scan-path. The synthesis tool such as ATPG creates during the synthesis a catalogue of undetected faults. The test coverage can thus be estimated and defines the lower limit of the achieved test coverage with the applied test pattern. It is important to notice that the test coverage is limited to the circuit logic, which

is covered by the scan-path. The modules such as memory, BIST or part of circuits that are not integrated in scan-path are not considered in the estimation of test coverage.

E.34 Justification of proven-in-use for applied hard cores

Aim: Avoidance of systematic failure during the application of hard cores

Description: A hard core is generally regarded as a black box representing the desired functionality and is composed of layout data basis in target technology that provides the desired circuit component. The possible functional failure can be treated in analogy to discrete components like, standard microprocessors, memories etc. The operation of such hard cores without the verification of correct functionality is possible, if for the applied target technology the used core can be considered as proven-in-use component. The rest of the circuit should then be verified intensively.

E.35 Application of validated hard cores

NOTE See also E.6 "Functional test on module level".

Aim: Avoidance of systematic failure during the application of hard cores.

Description: The core validation should be carried out by vendors, because of the complex nature of the core and assumed constraints, during the design phase on the basis of the (V)HDL source code. The validation can be justified only for the configuration and the target technology of the applied component.

E.36 On-line testing of hard cores

NOTE See also E.13 "Coverage of the verification scenarios (test benches)".

Aim: Avoidance of systematic failure during the application of hard cores.

Description: Verification of correct function and implementation of used hard cores by application of on-line tests. In applying this measure an efficient test concept is necessary and the evaluation of the applied concept should be documented.

E.37 Design rule check (DRC)

Aim: Verification of vendor design rules.

Description: Verification of the generated layout with respect to vendor design rules, for example minimum wire lengths, maximum wire lengths and several rules regarding placement of layout structures. A complete and correct run of DRC should be documented in detail.

E.38 Verification of layout versus schematic (LVS)

Aim: Independent verification of the layout.

Description: LVS extracts the circuit functionality from the layout data basis and compares the extracted circuit elements including interconnections with the input netlist. This assures the equivalence of circuit layout with the netlist specifying the circuit functionality. A complete and correct run of LVS should be documented in detail.

E.39 Additional slack (>20 %) for process technologies in use for less than 3 years

Aim: Assurance of the robustness of the implemented circuit functionality even under strong process and parameter fluctuation.

Description: The actual circuit behaviour is defined by number of overlapping physical effects particularly for small structures (for example below 0,5 µm). As a matter of fact, due to the lack of detail knowledge and necessary simplifications an exact model of circuit elements cannot be derived. With decreasing geometrical structures line delays play more and more dominant role. Signal delays along the wire and cross-coupling capacities between the wires grow over proportional. Signal delays are no longer negligible compared to gate delays. Estimated line delays depict increasing risk with decreasing geometrical structures.

Therefore it is recommended to plan an adequate amount of slack (> 20 %) with respect to minimal and maximal timing constraints for circuits designed using processes in use for less than 3 years, in order to guarantee correct operation of the circuit functionality in presence of strongly fluctuating parameters during the production or due to lack of precise modelling.

E.40 Burn-in Test

Aim: Assurance of the robustness of the manufactured chip. Weed out early failures. Bare die chip products do not have to prove their robustness by burn-in but, e.g., by wafer-level stress methods.

Description: The burn-in test should be carried out at the highest tolerable operating temperature (generally 125 °C). The test duration is depending on the aimed SIL-Level or on specific burn-in recommendations for example of the ASIC manufacturer. Burn in can be used to:

– weed out early failures (beginning of bathtube curve with decreasing failure rate);
– prove that early failures are already weeded out during manufacturing and testing (i.e. that devices out of the production line are already in the region of constant failure rate of the bathtube curve).

E.41 Application of proven-in-use device-series

Aim: Assurance of the reliability of the manufactured chips.

Description: The manufacturer of a safety design should have sufficient application experience with the used programmable device technology and the concerning developing tools.

E.42 Proven-in-use production process

Aim: Assurance of the reliability of the manufactured chips.

Description: A proven-in-use production process is characterised by a sufficient series production experience.

E.43 Quality control of the production process

Quality measures and control mechanisms during the device production process ensure a continuous process control. For example optical or electrical control of test structures,

temperature humidity bias-tests or temperature cycle test (see IEC 60068-2-1, IEC 60068-2-2 etc.).

E.44 Manufacturing quality pass of the device

The device quality will be proved by carrying out selected part-stress tests, for example temperature humidity bias-tests or change of temperature tests (see IEC 60068-2-1, IEC 60068-2-2 etc.). The device manufacturer will give proof of it.

E.45 Functional quality pass of the device

All devices will be functionally tested. The device manufacturer will give proof of it.

E.46 Quality standards

The ASIC manufacturer should provide for a sufficient quality management, for example documented within a Quality & Reliability Handbook: ISO 9000 certification or SSQA-, Standard Supplier Quality Assessment

Annex F
(informative)

Definitions of properties of software lifecycle phases

Table F.1 – Software Safety Requirements Specification

(see IEC 61508-3 7.2 and IEC 61508-3 Table C.1)

	Property	Definition
1.1	Completeness with respect to the safety needs to be addressed by software	The Software Safety Requirements Specification addresses all the safety needs and constraints resulting from earlier phases of the safety lifecycle and allocated to the Software. Safety needs and constraints are usually stated in the inputs to the Software Safety Requirements Specification activity. This may include the specification of what the Software must not do or must avoid.
1.2	Correctness with respect to the safety needs to be addressed by software	The Software Safety Requirements Specification providing an appropriate answer to the safety needs and constraints assigned to the Software. The objective is to assure that what is specified will really guarantee safety in all the necessary conditions.
1.3	Freedom from intrinsic specification faults, including freedom from ambiguity	Internal completeness and consistency of the Software Safety Requirements Specification: providing all necessary information for all the functions and situations that can be derived from its statements; expressing no contradicting or inconsistent statements. Contrary to completeness and consistency with respect to safety needs, internal completeness and consistency can be assessed based on the Software Safety Requirements Specification only
1.4	Understandability of safety requirements	The Software Safety Requirements Specification is fully understandable without excessive effort by all those who need to read it, even if they have not been involved earlier in the project, provided that they have the required knowledge. One important objective is to facilitate verification and, possibly, modifications.
1.5	Freedom from adverse interference of non-safety functions with the safety needs to be addressed by software	The Software Safety Requirements Specification avoids requirements that are not necessary to safety of the EUC. The objective is to avoid unnecessary complexity in the design and implementation of the software, so as to reduce the risk of faults and of functions not important to safety interfering with, or jeopardizing, those that are important to safety
1.6	Capability of providing a basis for verification and validation	The Software Safety Requirements Specification gives rise to tests and examinations that generate objective evidence that the software satisfies the Software Safety Requirements Specification.

Table F.2 – Software design and development: software architecture design

(see IEC 61508-3 7.4.3 and IEC 61508-3 Table C.2)

	Property	Definition
2.1	Completeness with respect to Software Safety Requirements Specification	The Software Architecture Design addresses all the safety needs and constraints raised by the Software Safety Requirements Specification.
2.2	Correctness with respect to Software Safety Requirements Specification	The Software Architecture Design provides an appropriate answer to the specified software safety requirements.
2.3	Freedom from intrinsic design faults	The Software Architecture Design and the Design Documentation are free from faults that can be identified independently of any specified software Safety Requirement. Examples: deadlocks, access to unauthorised resources, resource leaks, intrinsic incompleteness (i.e., failure to address all the situations that derive from the design itself).
2.4	Simplicity and understandability. Predictability of behaviour.	The Software Architecture Design allowing a correct and accurate prediction, in all specified situations, of the functioning of the Software. In particular, these situations include erroneous and failure situations. Predictability implies in particular that the functioning does not depend on items that cannot be controlled by designers or users.
2.5	Verifiable and testable design	The Software Architecture Design and the Design Documentation allow and facilitate the production of credible evidence that all the specified software safety requirements are correctly taken into account by the Design and that the Design is free from intrinsic faults. Verifiability may imply derived properties like simplicity, modularity, clarity, testability, provability, etc., depending on the verification techniques used.
2.6	Fault tolerance	The Software Architecture Design gives assurance that the software will have a safe behaviour in the presence of errors (internal errors, errors of operators or of external systems). Defensive design may be active or passive. Active defensive designs may include, features like detection, reporting and containment of errors, graceful degradation and cleaning up of any undesirable side effects prior to the resumption of normal operation. Passive defensive designs include features that guarantee the imperviousness to particular types of errors or particular conditions (avalanches of inputs, particular dates and times) without the software taking any specific action.
2.7	Defence against Common Cause Failure from external events	The Software Architecture Design facilitates the identification of common cause failure modes and effective precautions against failure.

Table F.3 – Software design and development: support tools and programming language

(see IEC 61508-3 7.4.4 and IEC 61508-3 Table C.3)

	Property	Definition
3.1	Support the production of software with the required software properties	Means to provide detection of errors or the elimination of error prone constructs
3.2	Clarity of the operation and functionality of the tool	The provision of comprehensive coverage and feedback on all aspects of operation of the tool
3.3	Correctness and repeatability of output	The consistency and accuracy of the tool output for any given input

Table F.4 – Software design and development: detailed design

(see IEC 61508-3 7.4.5 and IEC 61508-3 7.4.6 and IEC 61508-3 Table C.4)

	Property	Definition
4.1	Completeness with respect to Software Safety Requirements Specification	Methods of detailed software design and production are adopted that ensure that the resulting software addresses all the safety needs and constraints assigned to the Software.
4.2	Correctness with respect to Software Safety Requirements Specification	There exists specific evidence to claim that the safety requirements assigned to the Software have been met by the developed software.
4.3	Freedom from intrinsic design faults	The developed software is free from intrinsic faults. Examples: deadlocks, access to unauthorised resources, resource leaks.
4.4	Simplicity and understandability Predictability of behaviour	The behaviour of the developed software is predictable by objective and convincing testing and analysis.
4.5	Verifiable and testable design	The developed software is verifiable and testable.
4.6	Fault tolerance / Fault detection	Techniques and designs give assurance that the developed software will behave safely in the presence of errors.
4.7	Freedom from common cause failure	Techniques and designs identify common cause failure modes and provide effective precautions against software failure.

Table F.5 – Software design and development: software module testing and integration

(see IEC 61508-3 7.4.7 and IEC 61508-3 7.4.8 and IEC 61508-3 Table C.5)

	Property	Definition
5.1	Completeness of testing and integration with respect to the design specifications	The software testing examines the software behaviour sufficiently thoroughly to ensure that all the requirements of the Software Design Specification have been addressed.
5.2	Correctness of testing and integration with respect to the design specifications (succesful completion)	The module testing task is completed, and there exists specific evidence to claim that the safety requirements have been met.
5.3	Repeatability	Consistent results are produced on repeating the individual assessments carried out as part of the module testing and integration.
5.4	Precisely defined testing configuration	The module testing and integration has been applied to the right version of the elements and the software, with the results claimed, and allows the results to be linked to the specific configuration of the “as-built” software.

Table F.6 – Programmable electronics integration (hardware and software)

(see IEC 61508-3 7.5 and IEC 61508-3 Table C.6)

	Property	Definition
6.1	Completeness of integration with respect to the design specifications	The integration provides the appropriate depth and coverage of the system elements to demonstrate that it can perform the intended functions and does not perform unintended functions under all foreseeable operating conditions and under system failure. This covers the principles used for the verification, the targeted levels of design and aspects of the integration (for example verification of completeness of interaction between modules)
6.2	Correctness of integration with respect to the design specifications (successful completion)	The integration is based on correct assumptions. E.g., correctness of expected results, of the conditions of use considered, representativeness of test environments. The integration task is completed, and there exists specific evidence to claim that the safety requirements have been met.
6.3	Repeatability	Consistent results are produced on repeating the individual assessments carried out as part of the integration.
6.4	Precisely defined integration configuration	The integration gives appropriate assurance that it has been effectively applied as documented, to the right version of the elements and the Software, with the results claimed, and allows the results to be linked to the specific configuration of the “as-built” Software.

Table F.7 – Software aspects of system safety validation

(see IEC 61508-3 7.7 and IEC 61508-3 Table C.7)

	Property	Definition
7.1	Completeness of validation with respect to the Software Design Specification	The software validation addresses all the requirements of the Software Design Specification.
7.2	Correctness of validation with respect to the Software Design Specification (successful completion)	The software validation task is completed, and there exists specific evidence to claim that the safety requirements have been met.
7.3	Repeatability	Consistent results are produced on repeating the individual assessments carried out as part of the software validation.
7.4	Precisely defined validation configuration	The clear and concise definition of the system the requirements the environment

Table F.8 – Software modification

(see IEC 61508-3 7.8 and IEC 61508-3 Table C.8)

	Property	Definition
8.1	Completeness of modification with respect to its requirements	The modification has been properly approved by authorised personnel, with an appropriate understanding of its functional, safety, technical and operational consequences.
8.2	Correctness of modification with respect to its requirements	The modification achieves its specified objectives.
8.3	Freedom from introduction of intrinsic design faults	The modification does not introduce new systematic faults. Examples: division by zero, out of bound indexes or pointers, use of non initialised variables.
8.4	Avoidance of unwanted behaviour	The modification does not introduce any behaviour that, according to constraints stated in the Software Safety Requirements Specification, must be avoided.
8.5	Verifiable and testable design	The software design is such that the effect of the modification is capable of being examined thoroughly.
8.6	Regression testing and verification coverage	The software design is such that effective and thorough regression testing is possible to demonstrate that the software after modification continues to satisfy the Software Safety Requirements Specification.

Table F.9 – Software verification

(see IEC 61508-3 7.9 and IEC 61508-3 Table C.9)

	Property	Definition
9.1	Completeness of verification with respect to the previous phase	The verification is capable of establishing that the software satisfies all relevant requirements of the Software Safety Requirements Specification.
9.2	Correctness of verification with respect to the previous phase (successful completion)	The verification task is completed, and there exists specific evidence to claim that the safety requirements have been met.
9.3	Repeatability	Consistent results are produced on repeating the individual assessments carried out as part of the verification.
9.4	Precisely defined verification configuration;	The verification has been applied to the right version of the elements and the Software, with the results claimed, and allows the results to be linked to the specific configuration of the “as-built” Software.

Table F.10 – Functional safety assessment

(see IEC 61508-3 Clause 8 and IEC 61508-3 Table C.10)

	Property	Definition
10.1	Completeness of functional safety assessment with respect to this standard	The software functional safety assessment produces a clear statement on the extent of compliance found, the judgements made, remedial actions and timescales recommended, the conclusions reached and the recommendations arising for acceptance, qualified acceptance, or rejection and for any time constraints placed on these recommendations.
10.2	Correctness of software functional safety assessment with respect to the Design Specifications (succesful completion)	The software functional safety assessment task is completed, and there exists specific evidence to claim that the safety requirements have been met.
10.3	Traceable closure of all identified issues	There is a clear statement on the extent to which the issues arising during software functional safety assessment have been addressed.
10.4	The ability to modify the software functional safety assessment after change without the need for extensive re-work of the assessment	The software functional safety assessment is capable of being reworked to allow parts of the software functional safety assessment to be re-assessed after software change and for revised conclusions to be achieved, without the need for extensive rework of the complete software functional safety assessment.
10.5	Repeatability	The functional safety assessment is carried out against a consistent, planned and open process on identified individuals and document, which allows scrutiny of the basis for the assessments and the judgement achieved to all those affected by its judgements including system providers, users, maintainers and regulators. The functional safety assessment allows independent competent personnel to repeat the individual assessments carried out as part of the assessment.
10.6	Timeliness	The functional safety assessment is carried out at an appropriate frequency linked to the software safety lifecycle phases and at least prior to determined hazards being present, and it provides timely reporting of deficiencies. The outcome of tests, inspections, analyses etc. are actually available when they are required as input to an assessment decision.
10.7	Precisely defined configuration	The software functional safety assessment allows the results to be linked to the specific configuration of the system which is to be substantiated by the functional safety assessment results.

Annex G
(informative)

Guidance for the development of safety-related object oriented software

All of the recommendations of this standard for software design apply to object oriented software. Because the object-oriented approach presents information differently from procedural or functional approaches, the following list contains those recommendations that need specific consideration:

– understanding class hierarchies, and identification of the software function(s) that will be executed upon the invocation of a given method (including when using an existing class library);
– structure-based testing (IEC 61508-3, Table B.2 and IEC 61508-7, C.5.8)

Tables G.1 and G.2 provide informative guidance for the use of object oriented software to supplement the more general normative guidance provided in IEC 61508-3 Tables A.2 and A.4.

Table G.1 – Object Oriented Software Architecture

	Recommendation	Details	SIL1	SIL2	SIL3	SIL4
G1.1	Traceability of the concept of the application domain to the classes of the architecture.	Note 1	R	HR	HR	HR
G1.2	Use of suitable frames, commonly used combinations of classes and design patterns. NOTE When using existing frames and design patterns, the requirements of pre-developed software apply to these frames and patterns.	Note 2	R	HR	HR	HR

NOTE 1 Traceability from application domain to class architecture is less important.

NOTE 2 EXAMPLE 1: For a part of the intended safety-related project a frame might exist from a non safety-related project that has successfully solved a similar task and that is well known to the project participants. Then use of that frame is recommended.

EXAMPLE 2: It may happen that different algorithms are needed for solving closely connected sub tasks of the safety-related project. The strategy pattern can be chosen for accessing the algorithms.

EXAMPLE 3: Part of the safety-related project may consist of issuing proper warnings to inner and outer stake holders. The observer pattern can be chosen for organizing these warnings. The requirement does not apply for libraries.

NOTE 3 It is usually an abstract basic class that provides access to the derived concrete classes.

Table G.2 – Object Oriented Detailed Design

	Recommendation	SIL1	SIL2	SIL3	SIL4
G2.1	Classes should have only one objective	R	R	HR	HR
G2.2	Inheritance used only if the derived class is a refinement of its basic class	HR	HR	HR	HR
G2.3	Depth of inheritance limited by coding standards	R	R	HR	HR
G2.4	Overriding of operations (methods) under strict control	R	HR	HR	HR
G2.5	Multiple inheritance used only for interface classes	HR	HR	HR	HR
G2.6	Inheritance from unknown classes			NR	NR
G2.7	Verification that the reused object oriented libraries meet the recommendations of this table	HR	HR	HR	HR

NOTE 1 In other words: One class is characterised by having one responsibility, i.e. taking care of closely connected data and the operations on these data.

NOTE 2 Care is required to avoid circular dependencies between objects.

The following terms used above are informally defined here.

Table G.3 – Some Oriented Detailed terms

Term	Informal definition
Basic class	Class that has derived classes. A Basic Class is sometimes called upper class or parent class.
Derived Class	Class (assembly of attributes and operations) that inherits attributes and/or operations from another class (basic class). A derived class is sometimes called subclass or child class.
Frame	Structure of a program, in many cases pre-developed in order to be filled out for the specific application
Overriding	Replacing an operation (method, subroutine) by another operation (method, subroutine) of the same signature and inheritance hierarchy during run-time; property of object-oriented languages or programs; implements polymorphism
Signature of an operation	Name of an operation (subroutine, method), together with its parameters (arguments) and their types, occasionally also their return types. Two signatures are equal if they have the same names, number and types of parameters; in some languages also the return types have to be equal.

Bibliography

[1] IEC 60068-1:1988, *Environmental testing – Part 1: General and guidance*

[2] IEC 60529:1989, *Degrees of protection provided by enclosures (IP Code)*

[3] IEC 60812:2006, *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*

[4] IEC 60880:2006, *Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions*

[5] IEC 61000-4-1:2006, *Electromagnetic compatibility (EMC) – Part 4-1: Testing and measurement techniques – Overview of IEC 61000-4 series*

[6] IEC 61000-4-5:2005, *Electromagnetic compatibility (EMC) – Part 4-5: Testing and measurement techniques – Surge immunity test*

[7] IEC/TR 61000-5-2:1997, *Electromagnetic compatibility (EMC) – Part 5: Installation and mitigation guidelines – Section 2: Earthing and cabling*

[8] IEC 61025:2006, *Fault tree analysis (FTA)*

[9] IEC 61069-5:1994, *Industrial-process measurement and control – Evaluation of system properties for the purpose of system assessment – Part 5: Assessment of system dependability*

[10] IEC 61078:2006, *Analysis techniques for dependability – Reliability block diagram and boolean methods*

[11] IEC 61131-3:2003, *Programmable controllers – Part 3: Programming languages*

[12] IEC 61160:2005, *Design review*

[13] IEC 61163-1:2006, *Reliability stress screening – Part 1: Repairable assemblies manufactured in lots*

[14] IEC 61164:2004, *Reliability growth – Statistical test and estimation methods*

[15] IEC 61165:2006, *Application of Markov techniques*

[16] IEC 61326-3-1:2008, *Electrical equipment for measurement, control and laboratory use – EMC requirements – Part 3-1: Immunity requirements for safety-related systems and for equipment intended to perform safety-related functions (functional safety) – General industrial applications*

[17] IEC 61326-3-2:2008, *Electrical equipment for measurement, control and laboratory use – EMC requirements – Part 3-2: Immunity requirements for safety-related systems and for equipment intended to perform safety-related functions (functional safety) – Industrial applications with specified electromagnetic environment*

[18] IEC 81346-1:2009, *Industrial systems, installations and equipment and industrial products – Structuring principles and reference designations – Part 1: Basic rules*

[19] IEC 61506:1997, *Industrial-process measurement and control – Documentation of application software*

[20] IEC/TR 61508-0:2005, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 0: Functional safety and IEC 61508*

[21] IEC 61511 (all parts), *Functional safety – Safety instrumented systems for the process industry sector*

[22] IEC 62061:2005, *Safety of machinery – Functional safety of safety-related electrical, electronic and programmable electronic control systems*

[23] IEC 62308:2006, *Equipment reliability – Reliability assessment methods*

[24] ISO/IEC 1539-1:2004, *Information technology – Programming languages – Fortran – Part 1: Base language*

[25] ISO 5807:1985, *Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*

[26] ISO/IEC 7185:1990, *Information technology – Programming languages – Pascal*

[27] ISO/IEC 8631:1989, *Information technology – Program constructs and conventions for their representation*

[28] ISO/IEC 8652:1995, *Information technology – Programming languages – Ada*

[29] ISO 8807:1989, *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*

[30] ISO/IEC 9899:1999, *Programming languages – C*

[31] ISO/IEC 10206:1991, *Information technology – Programming languages – Extended Pascal*

[32] ISO/IEC 10514-1:1996, *Information technology – Programming languages – Part 1: Modula-2, Base Language*

[33] ISO/IEC 10514-3:1998, *Information technology – Programming languages – Part 3: Object Oriented Modula-2*

[34] ISO/IEC 13817-1:1996, *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*

[35] ISO/IEC 14882:2003, *Programming languages – C++*

[36] ISO/IEC/TR 15942:2000, *Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems*

[37] IEC 61800-5-2, *Electromagnetic compatibility (EMC) – Part 5: Installation and mitigation guidelines – Section 2: Earthing and cabling*

[38] IEC 60601 (all parts), *Medical electrical equipment*

[39] IEC 60068-2-1, *Environmental testing – Part 2-1: Tests – Test A: Cold*

[40] IEC 60068-2-2, *Environmental testing – Part 2-2: Tests – Test B: Dry heat*

[41] ISO 9000*, Quality management systems – Fundamentals and vocabulary*

[42] IEC 61508-1:2010, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements*

[43] IEC 61508-2:2010, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems*

[44] IEC 61508-3:2010, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*

[45] IEC 61508-6: 2010, *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3*

Index