COMMISSION ELECTROTECHNIQUE INTERNATIONALE **CEI IEC 61508-7**

INTERNATIONAL ELECTROTECHNICAL COMMISSION

Functional safety of electrical/electronic/ programmable electronic safety-related systems

Part 7: Overview of techniques and measures

Contents

Tables

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

Part 7: Overview of techniques and measures

FOREWORD

1) The IEC (International Electrotechnical Commission) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC national committees). The object of the IEC is to promote international cooperation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, the IEC publishes international standards. 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. The 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 the IEC on technical matters, prepared by technical committees on which all the national committees having a special interest therein are represented, express, as nearly as possible, an international consensus of opinion on the subjects dealt with.

3) They have the form of recommendations for international use published in the form of standards, technical reports or guides and they are accepted by the national committees in that sense.

4) In order to promote international unification, IEC national committees undertake to apply IEC international standards transparently to the maximum extent possible in their national and regional standards. Any divergence between the IEC standard and the corresponding national or regional standard shall be clearly indicated in the latter.

5) Attention is drawn to the possibility that some of the elements of IEC 61508 may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

6) The IEC has not laid down any procedure concerning marking as an indication of approval and has no responsibility when an item of equipment is declared to comply with one of its standards.

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

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

FDIS	Report on voting
65A/xxx	65A/xxx

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

Annexes A, B, C and D are for information only.

IEC 61508 consists of the following parts, under the general title “Functional safety of electrical/ electronic/programmable electronic safety-related systems”:

— Part 1: General requirements;

— Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems;

— Part 3: Software requirements;

— Part 4: Definitions and abbreviations;

— Part 5: Examples of methods for the determination of safety integrity levels;

— Part 6: Guidelines on the application of parts 2 and 3;

— Part 7: Overview of techniques and measures.

Introduction

Systems comprised of electrical and/or electronic components have been used for many years to perform safety functions in most application sectors. Computer-based systems (generically referred to as programmable electronic systems (PESs)) 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 those decisions.

This standard sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic components (electrical/electronic/ programmable electronic systems (E/E/PESs)) 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 application sector standards.

In most situations, safety is achieved by a number of protective 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 standard is concerned with electrical/electronic/programmable electronic (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 recognised that there is a great variety of E/E/PES applications in a variety of application sectors and covering a wide range of complexity, hazard and risk potentials. In any particular application, the exact prescription of safety measures will be dependent on many factors specific to the application. This standard, by being generic, will enable such a prescription to be formulated in future application sector international standards.

This standard:

— considers all relevant overall, E/E/PES and software safety lifecycle phases (for example, from initial concept, through design, implementation, operation and maintenance to decommissioning) when E/E/PESs 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 application sector international standards, dealing with safety-related E/E/PESs, to be developed – the development of 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;

— uses 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;

— adopts a risk-based approach for the determination of the safety integrity level requirements;

— sets numerical target failure measures for E/E/PE safety-related systems which are linked to the safety integrity levels;

— sets a lower limit on the target failure measures, in a dangerous mode of failure, that can be claimed for 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 failure of 10^{-5} to perform its design function on demand,

  — a high demand or continuous mode of operation, the lower limit is set at a probability of a dangerous failure of 10^{-9} per hour;

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

— adopts a broad range of principles, techniques and measures to achieve functional safety for E/E/PE safety-related systems, but does not use the concept of fail safe, which may be of value when the failure modes are well defined and the level of complexity is relatively low – the concept of fail safe was considered inappropriate because of the full range of complexity of E/E/PE safety-related systems that are within the scope of the standard.

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 parts 2 and 3 of this international standard.

1.2 Parts 1, 2, 3 and 4 of this standard are basic safety publications, although this status does not apply in the context of low complexity E/E/PE safety-related systems (see 3.4.4 of part 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 ISO/IEC Guide 104 and ISO/IEC Guide 51. One of the responsibilities of a Technical Committee is, wherever applicable, to make use of basic safety publications in the preparation of its own publications. IEC 61508 is also intended for use as a stand-alone standard.

1.3 Figure 1 shows the overall framework for parts 1 to 7 of this standard and indicates the role that part 7 plays in the achievement of functional safety for E/E/PE safety-related systems.

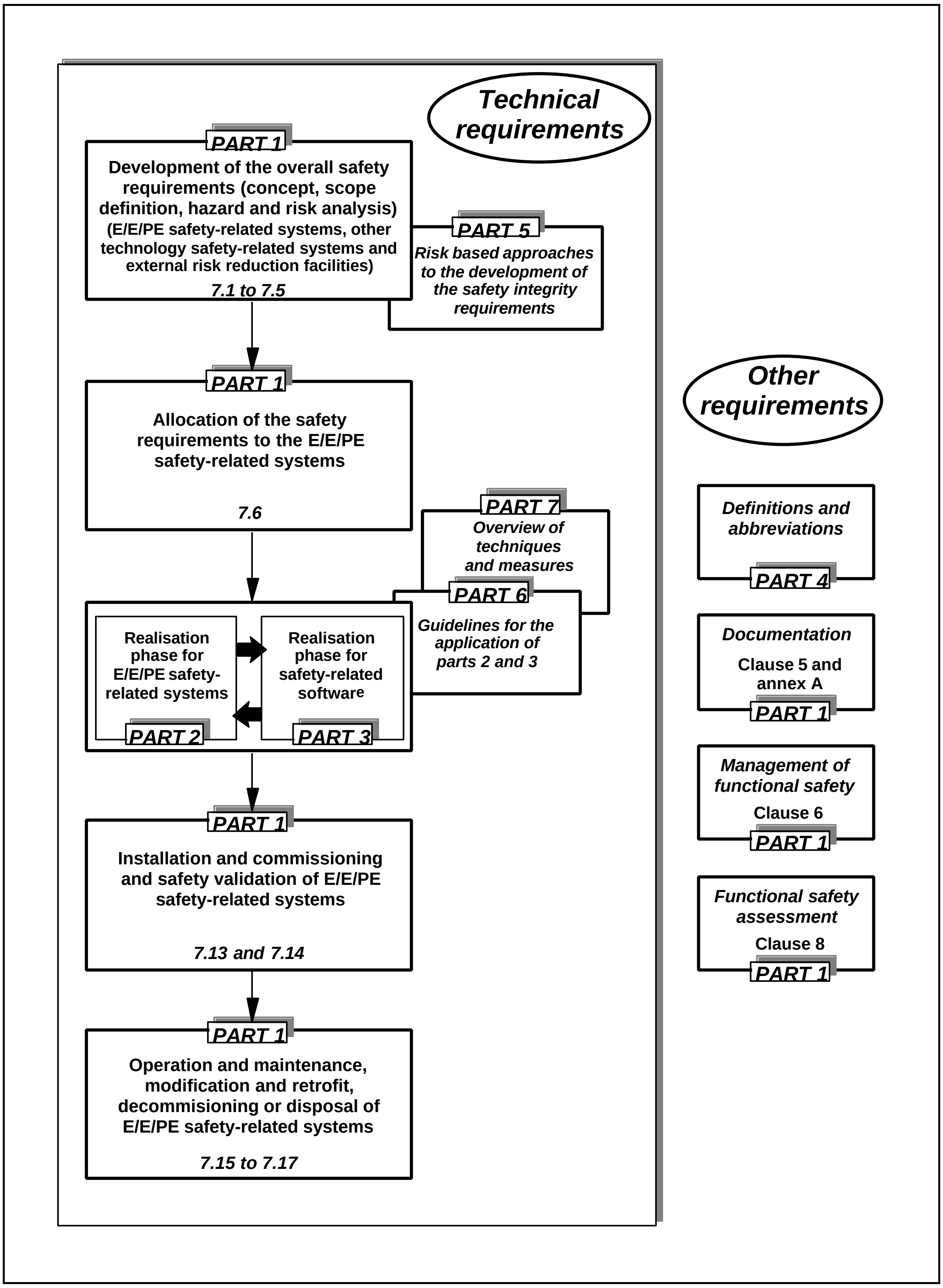
Technical requirements
PART 1
Development of the overall safety requirements (concept, scope definition, hazard and risk analysis)
(E/E/PE safety-related systems, other technology safety-related systems and external risk reduction facilities)
7.1 to 7.5
PART 5
Risk based approaches to the development of the safety integrity requirements
PART 1
Allocation of the safety requirements to the E/E/PE safety-related systems
7.6
PART 7
Overview of techniques and measures
PART 6
Guidelines for the application of parts 2 and 3
Realisation phase for E/E/PE safety-related systems
PART 2
Realisation phase for safety-related software
PART 3
PART 1
Installation and commissioning and safety validation of E/E/PE safety-related systems
7.13 and 7.14
PART 1
Operation and maintenance, modification and retrofit, decommisioning or disposal of E/E/PE safety-related systems
7.15 to 7.17
Other requirements
Definitions and abbreviations
PART 4
Documentation
Clause 5 and annex A
PART 1
Management of functional safety
Clause 6
PART 1
Functional safety assessment
Clause 8
PART 1

Figure 1 — Overall framework of this standard

2 Definitions and abbreviations

For the purposes of this standard, the definitions and abbreviations given in part 4 apply.

Annex A
(informative)

Overview of techniques and measures for E/E/PES: Control of random hardware failures (referenced by part 2)

A.1 Electrical

Global Objective: To control failures in electromechanical components.

A.1.1 Failure detection by on-line monitoring of equipment under control

NOTE This technique/measure is referenced in tables A.2, A.3, A.7 and A.14 to A.19 of part 2.

Aim: To detect failures which can be monitored by the equipment under control (EUC).

Description: Under certain conditions, failures can be detected using information about (for example) the time behaviour of the EUC. 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.15 of part 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 shutdown, or ensure that shutdown 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.

Anlagensicherung mit Mitteln der MSR-Technik. G Strohrman, Oldenburg, 1983.

A.1.3 Comparator

NOTE This technique/measure is referenced in tables A.2, A.3, A.4 of part 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 part 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.

Anlagensicherung mit Mitteln der MSR-Technik. Praxis der Sicherheitstechnik, Vol 1. Dechema 1988.

Sicherung von Anlagen der Verfahrenstechnik mit Mitteln der Mess-, Steuerungs- und Regelungstechnik. VDI/VDE Blatt 1 to 5, 1984 to 1988.

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

NOTE This technique/measure is referenced in tables A.2, A.9, A.14 and A.15 of part 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.

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.16, A.17 and A.19 of part 2.

Aim: To detect failures using hardware redundancy, ie 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.

Reference: DIN V VDE 0801: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben (Principles for Computers in Safety-Related Systems), Beuth-Verlag, Berlin, 1990.

A.2.2 Dynamic principles

NOTE This technique/measure is referenced in table A.3 of part 2.

Aim: To detect static failures by dynamic signal processing.

Description: An automatic change of 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.

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

NOTE This technique/measure is referenced in table A.3, A.16 and A.19 of part 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, comprising 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 1149.1-1990: Standard Test Access Port and Boundary-Scan Architecture.

A.2.4 Fail-safe hardware

NOTE This technique/measure is referenced in table A.3 of part 2.

Aim: To put a system into a safe state if a failure occurs.

Description: In hard-wired systems, a unit is said to operate in a fail-safe manner if:

— a defined set of faults will lead to a safe condition, and

— they are detected.

EXAMPLE The defined set of faults could include stuck-at faults, stuck-open, short circuits within and between components and directed short circuits.

References:

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

Elektronik in der Sicherheitstechnik. H Jürs, D Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch, Erich-Schmidt Verlag, Bielefeld 1993.

A.2.5 Monitored redundancy

NOTE This technique/measure is referenced in table A.3 of part 2.

Aim: To control 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 (ie if the output signals from all channels are not identical).

References:

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

Elektronik in der Sicherheitstechnik. H Jürs, D Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch, Erich-Schmidt Verlag, Bielefeld 1993.

A.2.6 Electrical/electronic components with automatic check

NOTE This technique/measure is referenced in table A.3 of part 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:

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

Elektronik in der Sicherheitstechnik. H Jürs, D Reinert, Sicherheitstechnisches Informations- und Arbeitsblatt 330220, BIA-Handbuch, Erich-Schmidt Verlag, Bielefeld 1993.

A.2.7 Analogue signal monitoring

NOTE This technique/measure is referenced in tables A.3 and A.14 of part 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 high level of confidence in the transmitter, eliminating the requirement to proof-test the transmitter function. External interfaces, for example impulse lines, may still require proof-testing.

Reference: UKOOA Guidelines for Instrument-Based Systems, UK Offshore Operators Association Limited, December 1995.

A.2.8 De-rating

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 part 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 data patterns (for example 55hex and AAhex).

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

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

NOTE This technique/measure is referenced in table A.4 of part 2.

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

Description: The hardware contains standard memory without any parity bit. 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 (registers) medium. However, the diagnostic coverage is only 90%.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

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

NOTE This technique/measure is referenced in table A.4 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.3.4 Coded processing (one channel)

NOTE This technique/measure is referenced in table A.4 of part 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:

The Coded Microprocessor Certification. P Ozello, Proc. SAFECOMP '92, 185-190, 1992.

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

Le Processeur Codé: un nouveau concept appliqué à la securité des systemes de transports. Gabriel, Martin, Wartski, Revue Generale des chemins de Ferm, No 6, June 1990.

A.3.5 Reciprocal comparison by software

NOTE This technique/measure is referenced in table A.4 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

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 This technique/measure is referenced in table A.5 of part 2.

Aim: To detect all single bit failures, all 2-bit failures, some 3-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, whether or not a corruption has taken place can be determined by checking the redundant bits. 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:

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

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

A.4.2 Modified checksum

NOTE This technique/measure is referenced in table A.5 of part 2.

Aim: To detect all odd-bit failures, ie 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.4.3 Signature of one word (8 bit)

NOTE This technique/measure is referenced in table A.5 of part 2.

Aim: To detect all 1-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.

References:

Calculating an error checking character in software. S Vasa, Computer Design, 5, 1976.

Berechnung von Fehlererkennungswahrscheinlichkeiten bei Signaturregistern. D Leisengang, Elektronische Rechenanlagen 24, H. 2, S. 55-61, 1982.

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

NOTE This technique/measure is referenced in table A.5 of part 2.

Aim: To detect all 1-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.

References:

Signaturanalyse in der Datenverarbeitung. D Leisengang, M Wagner, Elektronik 32, H. 21, S. 67-72, 1983.

Signaturregister für selbsttestende ICs. B Könemann, J Mucha, G Zwiehoff, Größtintegration / NTG-Fachtagung Baden-Baden, S. 109-112, April 1977.

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 part 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.

References

Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

Computers can now perform vital safety functions safely. Otto Berg von Linde, Railway Gazette International, Vol 135, No 11, 1979.

A.5 Variable memory ranges

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

A.5.1 RAM test “checkerboard” or “march”

NOTE This technique/measure is referenced in table A.6 of part 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 is 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.

References:

Memory testing. W G Fee, LSI Testing (Tutorial at the COMPCON 77 in San Francisco), IEEE Computer Society, W G Fee (Ed), 81-88, 1978.

Memory testing. P Rosenfield, Electronics and Power, H. 1, P. 26-31, 1979.

Halbleiterspeicher-Testfolgen. Th John, E Schaefer, Elektronikpraxis, H. 6, 18-26 and H. 7, 10-14, 1980.

A.5.2 RAM test “walkpath”

NOTE This technique/measure is referenced in table A.6 of part 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.

References:

Memory testing. W G Fee, LSI Testing (Tutorial at the COMPCON 77 in San Francisco), IEEE Computer Society, W G Fee (Ed), 81-88, 1978.

Techniques for testing the microprocessor family. W Barraclough, A Chiang, W Sohl, Proceedings of the IEEE, 64 (6), 943-950, 1976.

A.5.3 RAM test “galpat” or “transparent galpat”

NOTE This technique/measure is referenced in table A.6 of part 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 (ie 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.

References:

Entwurf von Selbsttestprogrammen für Mikrocomputer. E Maehle, Microcomputing. Berichte der Tagung III/79 des German Chapter of the ACM, W Remmele, H Schecher, (ed.), Stuttgart, Teubner, 204-216, 1979.

Periodischer Selbsttest einer mikroprocessorgesteuerten Sicherheitsschaltung. U Stinnesbek, Diplomarbeit am Institut für theoretische Elektrotechnik der RWTH Aachen 1980.

A.5.4 RAM test “Abraham”

NOTE This technique/measure is referenced in table A.6 of part 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 30n, 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 part 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.

Reference: Integrierte Digitalbausteine. K Reiß, H Liedl, W Spichall, Berlin, 1970.

A.5.6 RAM monitoring with a modified hamming code

NOTE This technique/measure is referenced in table A.6 of part 2.

Aim: To detect all odd-bit failures, all 2-bit failures, some 3-bit and some multi-bit failures.

Description: Every word of the 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, 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 word and its address.

References:

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

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

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 part 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.

References

Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

Computers can now perform vital safety functions safely. Otto Berg von Linde, Railway Gazette International, Vol 135, No 11, 1979.

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.14 and A.15 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.6.2 Code protection

NOTE This technique/measure is referenced in tables A.7, A.16, A.17 and A.19 of part 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 safety provides dataflow-dependent failure detection of the input and output units, based on information redundancy and/or time redundancy.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.6.3 Multi-channel parallel output

NOTE This technique/measure is referenced in table A.7 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.6.4 Monitored outputs

NOTE This technique/measure is referenced in table A.7 of part 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.

References:

Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

MSR-Schutzeinrichtungen. Anforderungen und Massnahmen zur gesicherten Funktion. DIN V 19251, February 1995.

A.6.5 Input comparison/voting

NOTE This technique/measure is referenced in tables A.7 and A.14 of part 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.

References:

Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

MSR-Schutzeinrichtungen. Anforderungen und Massnahmen zur gesicherten Funktion. DIN V 19251, February 1995.

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 part 2.

Aim: To detect all odd-bit failures, ie 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.7.2 Multi-bit hardware redundancy

NOTE This technique/measure is referenced in table A.8 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.7.3 Complete hardware redundancy

NOTE This technique/measure is referenced in table A.8 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.7.4 Inspection using test patterns

NOTE This technique/measure is referenced in table A.8 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.7.5 Transmission redundancy

NOTE This technique/measure is referenced in table A.8 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.7.6 Information redundancy

NOTE This technique/measure is referenced in table A.8 of part 2.

Aim: To detect failures in bus communication.

Description: It is usual to carry out the transfer in blocks, and to calculate the checksum of each block.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

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 part 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.

A.8.2 Voltage control (secondary)

NOTE This technique/measure is referenced in table A.9 of part 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 part 2.

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

Description: Over- or undervoltage 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.

A.9 Temporal and logical program sequence monitoring

NOTE This group of techniques and measures is referenced in tables A.16, A.17 and A.19 of part 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.12 of part 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 watchdog is not triggered at a fixed period, but a maximum interval is specified.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

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

NOTE This technique/measure is referenced in tables A.10 and A.12 of part 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 longer or shorter time than expected, emergency action is taken.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.9.3 Logical monitoring of program sequence

NOTE This technique/measure is referenced in tables A.10 and A.12 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

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

NOTE This technique/measure is referenced in tables A.10 and A.12 of part 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.

Reference: Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

A.9.5 Temporal monitoring with on-line check

NOTE This technique/measure is referenced in tables A.10 and A.12 of part 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.17 and A.19 of part 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

NOTE This technique/measure is referenced in table A.11 of part 2.

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/PES. Before the temperature leaves the specified range, emergency action is taken.

A.10.2 Fan control

NOTE This technique/measure is referenced in table A.11 of part 2.

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

NOTE This technique/measure is referenced in table A.11 of part 2.

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

NOTE This technique/measure is referenced in table A.11 of part 2.

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

NOTE This technique/measure is referenced in table A.11 of part 2.

Aim: To prevent overheating by forced-air cooling.

Description: The temperature is monitored and forced-air cooling is introduced if the temperature is outside of a specified range. 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.13 of part 2.

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

Description: Electrical energy lines are separated from the lines carrying the information. The electrical field which could induce voltage spikes on the information lines decreases with distance. Separation is also useful for information lines and low voltage input lines.

A.11.2 Spatial separation of multiple lines

NOTE This technique/measure is referenced in table A.13 of part 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.13, A.17 and A.19 of part 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.

References:

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

EMC for Product Designers. Tim Williams, Newnes, 1992, ISBN 0-7506-1264-9.

Grounding and Shielding Techniques in Instrumentation. John Wiley & Sons, New York, 1986.

Principles and Techniques of Electromagnetic Compatibility. C Christopoulos, CRC Press, 1995.

Gestaltung von Maschinensteuerungen unter Berücksichtigung der elektromagnetischen Verträglichkeit. F Börner, Sicherheitstechnisches Informations- und Arbeitsblatt 330260, BIA-Handbuch. 20. Lfg. V/93, Erich Schmidt Verlag, Bielefeld.

A.11.4 Antivalent signal transmission

NOTE This technique/measure is referenced in tables A.13 and A.17 of part 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.

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.14 of part 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.

A.12.2 Positive-activated switch

NOTE This technique/measure is referenced in table A.14 of part 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.15 of part 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 electrical interlocking in A.1.2). The redundancy introduced by this monitoring can be used to trigger emergency action.

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.

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993.

A.13.2 Cross-monitoring of multiple actuators

NOTE This technique/measure is referenced in table A.15 of part 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.

Annex B
(informative)

Overview of techniques and measures for E/E/PES: Avoidance of systematic failures (referenced by parts 2 and 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 part 2.

Aim: To avoid failures by adoption of an organizational 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 organizational model, especially for quality assurance (see standards such as the series ISO 9000 to ISO 9004 or similar) 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 organizational 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:

IEEE: Software Engineering Standards. IEEE/Wiley-Interscience, New York, 1987

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

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993.

B.1.2 Documentation

NOTE This technique/measure is referenced in tables B.1 to B.6 of part 2.

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, ie:

- 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 life cycle;
- 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:

65/210/FDIS. Documentation of software for process control systems and facilities (future IEC 61506, in preparation).

EWICS European Workshop on Industrial Computer Systems, TC 7: Safety Related Computers - Software Development and Systems Documentation. Verlag TÜV Rheinland, Köln, 1985.

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993.

Entwicklungstechnik sicherheitsverantwortlicher Software in der Eisenbahn-Signaltechnik. U Feucht, Informatik-Fachberichte 86, Springer Verlag, Berlin, 184-195, 1984.

Richtlinie zur Erstellung und Prüfung sicherheitsrelevanter Software. K Grimm, G Heiner, Informatik Fachberichte 86, Springer Verlag, Berlin, 277-288, 1984.

B.1.3 Separation of safety-related systems from non safety-related systems

NOTE This technique/measure is referenced in tables B.1 and B.2 of part 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.

B.1.4 Diverse hardware

NOTE This technique/measure is referenced in tables A.16, A.17 and A.19 of part 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 over voltage, 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.

B.2 E/E/PES safety 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.2 of part 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.

Reference: Structured Analysis and System Specification. T De Marco, Yourdon Press, Englewood Cliffs, 1979.

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 and B.2 of part 2.

Aim: To prove that the design meets its specification.

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

Starting from a mathematically formal specification, the design can be transformed by a series of step-wise refinements to a logic circuit design.

References:

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

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

B.2.3 Semi-formal methods

NOTE This technique/measure is referenced in tables B.1, B.2 and B.3 of part 2 and in tables A.1, A.2 and A.4 of part 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, ie 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 subsections.

B.2.3.2 Finite state machines/state transition diagrams

NOTE This technique/measure is referenced in tables B.5 and B.7 of part 3.

Aim: To model, 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. 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 specification or design expressed as a finite state machine can be checked for:

— completeness (the system must have an action and new state for every input in every state);

— consistency (only one state change is described for each state/input pair); and

— reachability (whether or not it is possible to get from one state to another by any sequence of inputs).

These are important properties for critical systems. Tools to support these checks are easily developed. 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.

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

B.2.3.3 Time Petri nets

NOTE This technique/measure is referenced in tables B.5 and B.7 of part 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 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.

References:

Petri nets: Properties, analysis and applications. T Murata, Proc. IEEE, 77 (4), 541-580, April 1989.

Safety analysis using Petri nets. N G Leveson, J L Stolzy, Proc. 15th Ann. Int. Symp on Fault-Tolerant Computing, 358-363, IEEE, 1985.

Using Petri nets for safety analysis of unmanned Metro system. M El Koursi, P Ozello, Proc SAFECOMP '92, 135-139, Pergamon Press, 1992.

Net theory and applications. W Brauer (ed), Lecture Notes in Computer Science, Vol 84, Springer Verlag 1980.

Petri net theory and modelling of systems. J L Peterson, Prentice Hall, 1981.

A tool for requirements specification and analysis of real time software based on timed Petri nets. S Bologna, F Pisacane, C Ghezzi, D Mandrioli, Proc. SAFECOMP 88, 9-11 Nov. 1988. Fulda Fed. Rep. of Germany 1988.

B.2.4 Computer aided specification tools

NOTE This technique/measure is referenced in tables B.1 and B.2 of part 2 and in tables A.1 and A.2 of part 3.

B.2.4.1 General

Aim: To use formal specification techniques to facilitate automatic inspection 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 life cycle. Specification tools can be classified according to the following subclauses.

B.2.4.2 Tools oriented towards no specific method

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 the user a great deal of freedom but also gives him little of the specialised support necessary when creating specifications. This makes familiarisation with the system more difficult.

Reference: Integrierte Rechnerunterstützung für Entwicklung, Projektmanagement und Produktverwaltung mit EPOS. R Lauber, P Lempp, Elektron. Rechenanlagen 27, Heft 2, 68-74, 1985

B.2.4.3 Model orientated procedure with hierarchical analysis

This method gives a functional representation of the desired system (structured analysis) at various levels of abstraction (degree of precision). 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.

Reference: Structured Analysis for Requirement Definition. D T Ross, K E Schomann jr, IEEE Trans. on SE, Jan 1977.

B.2.4.4 Entity models

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.

References:

PSL/PSA Computer-aided Technique for Structured Documentation and Analysis of Information Processing. D Teichroew, E A Hershey, IEEE Trans on SE, Jan 1977.

Computer Aided Software Development. D Teichroew, E A Hershley, Y Lamamoto, Beitrag in: Verfahren und Hilfsmittel fur Spezifikation und Entwurf von Prozeßauto-matisierungssystemen. Hommel (ed), Bericht KfK-PDV 154, Kernforschungszentrum Karlsruhe, 1978.

PCSL und ESPRESO - zwei Ansätze zur Formalisierung der Prozessrechner Softwarespezifikation. J Ludewig, GI-Fachtagung Prozessrechner 1981, Informatik-Fachberichte Bd. 39, Springer Verlag, Berlin, 1981.

B.2.4.5 Incentive and answer

The relationships between the objects of the system are illustrated in the arrangement plan incentive and answer. A simple and easily expanded language is used which contains language elements which represent objects, relationships, characteristics and structures.

References:

A Requirements Engineering Methodology for Real-time Processing Requirements. M W Alford, IEEE Trans on SE, Jan 1977.

The Specification System X-SPEX - Introduction and Experiences. G Dahll, J Lahti, Proc. SAFECOMP '83, 111-118.

B.2.5 Checklists

NOTE This technique/measure is referenced in tables B.1, B.2 and B.3 of part 2 and in tables A.10 and B.8 of part 3.

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

Description: A set of questions to be completed by the person performing the checklist. Many of the questions are of a general nature and the assessor must interpret them as seems most appropriate to the particular system being assessed. Checklists can be used for all phases of the overall, E/E/PES 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:

Software for Computers in the Safety Systems of Nuclear Power Stations. IEC 60880, 1986.

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993.

Programmable Electronic Systems (PES) in Safety Related Application. Health and Safety Executive, UK, 1987.

Dependability of Critical Computer Systems 2, F J Redmill, Elsevier Applied Science, 1989, ISBN 1-85166-381-9.

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

B.2.6 Inspection of the specification

NOTE This technique/measure is referenced in tables B.1 and B.2 of part 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 organizational measures are covered. This procedure has proved itself to be very effective in practice.

References:

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

IEC 61160: 1992, Formal Design Review.

B.3 E/E/PES 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.3 of part 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:

EWICS European Workshop on Industrial Computer Systems, TC 7: Safety Related Computers - Software Development and Systems Documentation. Verlag TÜV Rheinland, Köln, 1985.

Guidelines for Safe Automation of Chemical Processes. CCPS, AIChE, New York, 1993.

Deutsche Bundesbahn: Richtlinien-Entwürfe 42500 to 42550 für das Handbuch “Grundsätze zur technischen Zulassung in der Signal- und Nachrichtentechnik”. Bundesbahn-Zentralamt München, August 1987.

Richtlinie zur Erstellung und Prüfung sicherheitsrelevanter Software. K Grimm, G Heiner, Informatik Fachberichte 86, Springer Verlag, Berlin, 277-288, 1984.

B.3.2 Structured design

NOTE This technique/measure is referenced in tables B.1 and B.3 of part 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 and

— use of manufactured and tested circuit parts.

References:

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

Software Design for Real-time Systems. J E Cooling, Chapman and Hall 1991.

Essential Systems Analysis. StM McMenamin, F Palmer, Yourdon Inc, 1984.

The Use of Structured Methods in the Development of Large Software-Based Avionic Systems. D J Hatley, Proceedings DASC, Baltimore, 1984.

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

System Development. M Jackson, Prentice-Hall, 1983.

MASCOT 3 User Guide. MASCOT Users Forum, RSRE, Malvern, England, 1987.

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

Structured Analysis for Requirements Definition, D T Ross, K E Schoman Jr, IEEE Trans Software Eng, Vol SE-3, 6-15, 1977.

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

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

Structured Analysis and Design Technique - Application on Safety Systems. W Heins, Risk Assessment and Control Courseware, Module B1, chapter 11, Delft University of Technology, Safety Science Group, PO Box 5050, 2600 GB Delft, Netherlands, 1989.

B.3.3 Use of well tried components

NOTE This technique/measure is referenced in tables B.1 and B.3 of part 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 components (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 components in the event of a failure occurring.

References:

Reliability Testing for Industrial use. W T Greenwood, Computer, 10 (7), 26-30, 1977.

Independent Test Labs: Covent Emptor. E Rubinstein, IEEE Spectrum, 14 (6), 44-50, 1977.

Microcomputers in safety technique - an aid to orientation for developer and manufacturer. H Hölscher, J Rader, Verlag TÜV Rheinland, Köln, 1986.

IEC 61163-1: 1995, Reliability Stress Screening — Part 1: Reparable Items Manufactured in Lots.

Zuverlässigkeit elektronischer Komponenten. Bajenescu, Titu, VDE-Verlag, Berlin, 1985.

B.3.4 Modularisation

NOTE This technique/measure is referenced in tables B.1 and B.3 of part 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 (ie shared data, exchange of information) is minimised. The complexity of individual subsystems is also restricted.

References:

EWICS European Workshop on Industrial Computer Systems, TC 7: Safety Related Computers - Software Development and Systems Documentation. TÜV Rheinland, Köln, 1985.

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

Software Reliability - Principles and Practices. G J Myers, Wiley-Interscience, New York, 1976.

Software Design for Real-time Systems. J E Cooling, Chapman and Hall, 1991.

B.3.5 Computer aided design tools

NOTE This technique/measure is referenced in tables B.1 and B.3 of part 2 and in table A.4 of part 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.

References:

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

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

B.3.6 Simulation

NOTE This technique/measure is referenced in tables B.1, B.3 and B.6 of part 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 safety-related system circuit’s function 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.

References:

Proc. Working Conference on Prototyping. Namur Oct 1983, Budde et al, Springer Verlag, 1984.

Using an executable specification language for an information system. S Urban et al, IEEE Trans Software Engineering, Vol. SE-11 No 7, July 1985.

Verification and validation of Real-time Software. W J Quirk (ed), Springer Verlag, Berlin, 1985.

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag, 1993.

B.3.7 Inspection (reviews and analysis)

NOTE This technique/measure is referenced in tables B.1 and B.3 of part 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 walkthrough, the author is passive and the inspector is active during the inspection procedure.

References:

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

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

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag 1993.

B.3.8 Walkthrough

NOTE This technique/measure is referenced in tables B.1 and B.3 of part 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 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:

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

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag 1993.

B.4 E/E/PES 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.5 of part 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.

B.4.2 User friendliness

NOTE This technique/measure is referenced in table B.5 of part 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;

— any incorrect intervention remains harmless;

— the intervention facilities and indication facilities are designed according to ergonomic requirements;

— 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.5 of part 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.1 and B.5 of part 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.

B.4.5 Operation only by skilled operators

NOTE This technique/measure is referenced in tables B.1 and B.5 of part 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.

B.4.6 Protection against operator mistakes

NOTE This technique/measure is referenced in tables B.1 and B.5 of part 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 correct.

B.4.7 Protection against sabotage

NOTE This technique/measure is referenced in table B.5 of part 2.

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

Description: In addition to the measures already described, other organisational measures are necessary, for example operator supervision.

B.4.8 Modification protection

NOTE This technique/measure is referenced in table A.18 of part 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.18 and A.19 of part 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.

Reference: DIN V VDE 0801: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben (Principles for Computers in Safety-Related Systems). Beuth-Verlag, Berlin, 1990.

B.5 E/E/PES 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.4 and B.6 of part 2 and in tables A.5, A.6 and A.7 of part 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 to test the manufactured components in combination with other partner components of the same batch, in order to reveal common mode faults which would otherwise have remained masked.

References:

Functional Program Testing and Analysis. W E Howden, McGraw-Hill, 1987.

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

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.1, B.4 and B.6 of part 2 and in tables A.5, A.6 and A.7 of part 3.

Aim: To check the dynamic behaviour under real functional conditions. To reveal specification failures 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 system's internal structure 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 black box 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 following:

— 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:

Functional Testing and Analysis. W E Howden, McGraw-Hill Book Company, New York, 1987

Software Testing and Validation Techniques. E Miller, W E Howden, IEEE Computer Society, New York, 1978.

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

Methodik systematischen Testens von Software. K Grimm, 30 (4), 1988.

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag 1993.

B.5.3 Statistical testing

NOTE This technique/measure is referenced in tables B.1, B.4 and B.6 of part 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:

Software Testing via Environmental Simulation (CONTESSE Report). Available until December 1998 from: Ray Browne, CIID, DTI, 151 Buckingham Palace Road, London, SW1W 9SS, UK, 1994.

Aspects of Development and Verification of Reliable Process Computer Software. W Ehrenberger, IFAC-IFIP Conference Proceedings, 35-48, 1980.

Verification and validation of Real-time Software. W J Quirk (ed), Springer Verlag, Berlin, 1985.

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag, 1993.

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

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

B.5.4 Proven in use

NOTE 1 See also C.2.10 for a similar measure and annex D for a statistical approach, both in the context of software.

NOTE 2 This technique/measure is referenced in tables B.1, B.4 and B.6 of part 2.

Aim: To use field experience from different applications to prove that the safety-related system will work according to its specification.

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. In the rarest cases this measure will be sufficient in isolation to reach the necessary safety integrity level. 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 proven by use. 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 proven by use, since they have never performed their fault detection function.

For proven by use to apply, the following requirements must have been fulfilled:

— unchanged specification;

— 10 systems in different applications;

— 10^5 operating hours and at least 1 year of service history.

The proof is given 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;

— users and time of application;

— operating hours;

— procedures for the selection of the systems and applications procured to the proof;

— procedures for fault detection and fault registration as well as fault removal.

Reference: DIN V VDE 0801 A1: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben (Principles for Computers in Safety-Related Systems). Änderung 1 zu DIN V VDE 0801/01.90. Beuth-Verlag, Berlin, 1994.

B.6 E/E/PES safety validation

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

B.6.1 Functional testing under environmental conditions

NOTE This technique/measure is referenced in table B.6 of part 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 mentioned standards).

References:

IEC 61000-4-1, Electromagnetic Compatibility (EMC) – Part 4: Testing and Measurement Techniques – Section 1: Overview of Immunity Tests.

IEC 60068-1, Environmental Testing – Part 1: General and Guidance.

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.1 and B.6 of part 2.

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

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:

Guide for surge withstand capability (SWC) test. ANSI C.37.90-1974.

IEC 61000-4-5, Electromagnetic compatibility (EMC) – Part 4: Testing and measurement techniques – Section 5: Surge immunity testing.

B.6.3 Calculation of failure rates

NOTE This technique/measure is referenced in tables B.1 and B.6 of part 2.

Aim: To estimate the lifetime of the facilities and the individual physical units, so that the necessary maintenance and inspection rates may be determined, and weak points in the unit detected.

Description: The calculation of the failure rates for the physical units of a subcircuit or a complete circuit requires the basic failure rates of components – as measured, for example, by the MIL handbooks or company standards – and the operating conditions of the circuits.

References:

Military Standardization Handbook 217 F: Reliability Prediction of Electronic Equipment. Rome Air Development Center, ATTN: RBRT, Griffiss Air Force Base, New York.

Evaluation control systems reliability. W Goble, 1992.

Reliability by Design. A C Brombacher, John Wiley & Sons 1992.

B.6.4 Static analysis

NOTE This technique/measure is referenced in tables B.1 and B.6 of part 2 and in table A.9 of part 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:

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

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag 1993.

B.6.5 Dynamic analysis

NOTE This technique/measure is referenced in tables B.1 and B.6 of part 2 and in tables A.5 and A.9 of part 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:

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

VDI-Gemeinschaftsausschuß Industrielle Systemtechnik: Software-Zuverlässigkeit. VDI-Verlag 1993.

B.6.6 Failure analysis

NOTE This technique/measure is referenced in tables B.1 and B.6 of part 2.

B.6.6.1 Failure modes and effects analysis

Aim: To analyse a system design, by examining 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, detection procedures and recommendations. If the recommendations are acted upon, they are documented as remedial action taken.

References:

IEC 60812: Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA), 1985.

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

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

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

B.6.6.2 Cause consequence diagrams

NOTE This technique/measure is referenced in tables A.10, B.3 and B.4 of part 3.

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

Description: The technique can be regarded as a combination of fault tree and event tree analysis. Starting from a critical event, a cause consequence graph is traced backwards and forwards. In the backwards direction it is equivalent to a fault tree with the critical event as the given top event. In the forward direction the possible consequences arising from an event are 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 to compute the probability of occurrence of certain critical consequences.

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

B.6.6.3 Event tree analysis

NOTE This technique/measure is referenced in tables A.10 and B.4 of part 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.

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. 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.

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

B.6.6.4 Failure modes, effects and criticality analysis

NOTE This technique/measure is referenced in tables A.10 and B.4 of part 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: Criticality 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.

Reference: Design Analysis Procedure for Failure Modes, Effects and Criticality Analysis (FMECA). Aerospace Recommended Practice (ARP) 926, Society of Automotive Engineers (SAE), USA, 15 September 1967.

B.6.6.5 Fault tree analysis

NOTE This technique/measure is referenced in tables A.10 and B.4 of part 3.

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

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

The method is graphical, and a set of standardised symbols are used to draw the fault tree. 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.

References:

IEC 61025: Fault Tree Analysis (FTA), 1990.

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

Fault Tree Handbook. W E Vesely et al, NUREG-0492, Division of System Safety Office of Nuclear Reactor Regulation, US Nuclear Regulatory Commission Washington, DC 20555, 1981.

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

B.6.7 Worst-case analysis

NOTE This technique/measure is referenced in tables B.1 and B.6 of part 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.1 and B.6 of part 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:

Functional Program Testing and Analysis. W E Howden, McGraw-Hill, 1987.

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

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.1 and B.6 of part 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.

B.6.10 Fault insertion testing

NOTE: This technique/measure is referenced in tables B.1 and B.6 of part 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 I and II 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 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:

Integrity Testing of Process Control Systems. R. J. Lasher, Control Engineering, 36 (11), 152-164, October 1989.

IEC 61069-5: Industrial process measurement and control - Evaluation of system properties for the purpose of system assessment - Part 5: Assessment of system dependability. 1st edition, 1994.

Annex C
(informative)

Overview of techniques and measures for achieving software safety integrity (referenced by part 3)

C.1 General

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

Some general references are:

System - Safety Society of America System Safety Analysis Handbook. System Safety Society, New Mexico Chapter. PO Box 95424, Alburquerque NM, USA.

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

Encyclopaedia of Software Engineering. Ed J Marciniak. John Wiley & Sons, ISBN 0-471-54004-8, 1994.

Software Engineer’s Reference Book. Ed J McDermid. Butterworth-Heinemann, ISBN 0-7506-1040-9, 1991.

C.2 Requirements and detailed design

NOTE Relevant techniques and measures may also be found in B.2.

C.2.1 Structured methods

NOTE This technique/measure is referenced in tables A.2 and A.4 of part 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 life-cycle. 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 exist. Some are designed for traditional data-processing and transaction processing functions, while others (MASCOT, JSD, real-time Yourdon) are more oriented to process control and real-time applications (which tend to be more safety critical).

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

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

— 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, ie lists of the sort of things that need analysis;

— low intellectual overhead - simple, intuitive, pragmatic.

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, JSD makes use of regular expressions; Yourdon, SOM and SDL utilise 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 five structured methods in more detail: Controlled Requirements Expression, Jackson System Development, MASCOT, real-time Yourdon, and Structured Analysis and Design Technique (SADT).

References:

Software Design for Real-time Systems. J E Cooling, Chapman and Hall, 1991.

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

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

The Use of Structured Methods in the Development of Large Software-Based Avionic Systems. D J Hatley. Proc. DASC, Baltimore, 1984.

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 Design for Real-time Systems. J E Cooling, Chapman and Hall, 1991.

C.2.1.3 JSD - Jackson System Development

Aim: A development method covering the development of software systems from requirements through to code, with special emphasis on real-time systems.

Description: JSD is a staged development 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:

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

System Development. M Jackson. Prentice-Hall, 1983.

C.2.1.4 MASCOT

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

Description: MASCOT (Modular Approach to Software Construction, Operation and Test) is a design method supported by a programming system. It is a systematic method of expressing the structure of real-time systems in a way that is independent of the target hardware or implementation language. It imposes a disciplined approach to design that yields a highly modular structure, ensuring a close correspondence between the functional elements in the design and the construction elements appearing in system integration. A system is designed in terms of a network of concurrent processes that communicate through channels. Channels can be either pools of fixed data or queues (pipelines of data). Control of access to channels is described independently of the processes in terms of access mechanisms that also enforce scheduling rules on the processes. Recent versions of MASCOT have been designed with Ada implementation in mind.

MASCOT supports an acceptance strategy based on the test and verification of single software modules and larger collections of functionally related software modules. A MASCOT implementation is intended to be built upon a MASCOT kernel - a set of scheduling primitives that underline the implementation and support the access mechanisms.

Reference: MASCOT 3 User Guide. MASCOT Users Forum. RSRE, Malvern, England, 1987.

C.2.1.5 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 the system carries out in response to events and a description of the data the system must hold in order to respond.

The implementation model also divides into sub-models, 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, 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:

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

Strategies for Real-time System Specification. D Hatley, E Pirbhai, Dorset Publishing House, 1988.

C.2.1.6 SADT - Structured Analysis and Design Technique

Aim: To model and analyse, in a diagrammatic form using information flows, the decision making processes and the management tasks associated with a complex system.

Description: In SADT, the concept of an activity-factor diagram plays a central role. An A/F diagram consists of activities grouped in so called 'action boxes'. Each action box has a unique name, and is linked to other action boxes by factor relations (drawn as arrows) which are also given unique names. Each action box can be hierarchically decomposed into subsidiary action boxes and relations. There are four types of factors: inputs, controls, mechanisms and outputs:

— **input:** indicated by an arrow that enters an action box at the left hand side. inputs can represent material or immaterial things and they are suitable for manipulation by one or more activities in an action box;

— **control:** typically an instruction, procedure, choice criterion or so on. a control guides the execution of an activity and is shown by an arrow entering the top side of an action box;

— **mechanism:** a resource such as a personnel, organisational units or equipment, needed for an activity to perform its task;

— **output:** anything that an activity produces, pictured by an arrow leaving an action box at the right hand side.

When activities are strongly related to each other by many factor relations, it is perhaps better to consider these activities as an indivisible group, contained in one action box, with no further detailing of its content. The guiding principle for grouping of activities into action boxes is that the resulting boxes are coupled pairwise by only a few factors.

The model hierarchy of A/F diagrams is pursued until a further detailing of the action boxes is meaningless. This stage is reached when the activities within the boxes are inseparable or when further detailing of the action boxes falls outside the scope of the system analysis.

References:

Structured Analysis for Requirements Definition. D T Ross, K E Schoman Jr. IEEE Transactions on Software Engineering, Vol SE-3, 1, 6-15, 1977.

Structured Analysis (SA): A Language for Communicating Ideas. D T Ross. IEEE Transactions on Software Engineering, Vol SE-3, 1, 16-34, 1977.

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

Structured Analysis and Design Technique - Application on Safety Systems. W Heins. Risk Assessment and Control Courseware, Module B1, Chapter 11, Delft University of Technology, Safety Science Group, PO Box 5050, 2600 GB Delft, Netherlands, 1989.

C.2.2 Data flow diagrams

NOTE This technique/measure is referenced in tables B.5 and B.7 of part 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 are made up of three components:

— 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 principle 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.2.1.5).

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.

Reference: Software Engineering. I Sommerville, Addison-Wesley, 3rd Edition, 1989.

C.2.3 Structure diagrams

NOTE This technique/measure is referenced in table B.5 of part 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 units without including any information about the order of activation of these units. They are drawn using the following three symbols:

— a rectangle annotated with the name of the unit;

— an arrow connecting these rectangles;

— a circled arrow, annotated with the name of data passed to and from elements in the structure chart – normally, the circled arrow is drawn parallel to the arrow connecting the rectangles in the chart.

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

Structure charts derived from data flow diagrams represent a first level structure of the system, where each box on the structure chart represents a bubble in the data flow diagram. Naturally, deeper levels can be described using the same technique.

References:

Software Engineering. I Sommerville, Addison-Wesley, 3rd Edition, 1989.

Structured Design. L L Constantine and E Yourdon, Englewood Cliffs, New Jersey, Prentice Hall, 1979.

Reliable Software Through Composite Design. G J Myers, New York, Van Nostrand, 1975.

C.2.4 Formal methods

NOTE This technique/measure is referenced in tables A.1, A.2, A.4 and B.5 of part 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.

NOTE The above description may also be found in B.2.2.

Several formal methods are described in the following subsections of this overview - CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z. Note that other techniques, such as finite state machines (see B.2.3.2) and Petri nets (see B.2.3.3), may be considered as formal methods, depending on how strictly the techniques, as used, conform to a rigorous mathematical basis.

References:

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 CCS - Calculus of Communicating Systems

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

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

References:

Communication and Concurrency. R Milner, Prentice-Hall, 1989.

A Calculus of Communicating Systems. R Milner, Report No ECS-LFCS-86-7, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK.

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

C.2.4.3 CSP - Communicating Sequential Processe s

Aim: CSP is a technique for the specification of concurrent software systems, ie. 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 -a permissible sequences 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. C A R Hoare, Prentice-Hall, 1985.

C.2.4.4 HOL - Higher Order Logic

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 LCF (logic of computable functions) system.

References:

HOL: A Machine Orientated Formulation of Higher Order Logic. M Gordon, University of Cambridge Technical Report, No 68.

HOL: A proof generating system for high order logic. M J C Gordon, VLSI Specification, Verification and Synthesis, (G Birtwistle and P A Subramanyam, eds), 73-128, 1988.

Application of formal methods to the VIPER microprocessor. W J Cullyer, C H Pygott, Proc IEEE, 134, 133-141, 1987.

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 a second component based on the abstract data type language ACT ONE. The process description component of LOTOS could, however, be used with other formalisms for the description of abstract data types.

Reference: ISO 8807: Information Processing Systems - Open Systems Inter-connection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, 1988.

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:

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

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

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

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

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 component or the computer program.

Quantified time intervals and constraints are not handled explicitly in temporal logic. Absolute timing has to be handled by creating additional time states as part of the state description.

References:

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

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

Logics for Computer Programming. D Gabay. Ellis Horwood.

C.2.4.8 VDM - Vienna Development Method

Aim: The systematic specification and implementation of sequential programs.

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

The specification technique is model-based in that the system state is modelled in terms of set-theoretic structures on which are 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 a program in the target language. Reification and refinement steps give rise to proof obligations that establish their correctness. Whether or not these obligations are carried out is a choice made by the designer.

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

References:

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

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

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

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

VDM-SL Reference Guide. J Dawes. Pitman Publishing, ISBN 0-273-03151-1, 1991.

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:

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

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

The B Method. J R Abrial et al, VDM '91 Formal Software Development Methods, (S Prehen and W J Toetenel, eds), Springer Verlag, 398-405, 1991.

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

C.2.5 Defensive programming

NOTE This technique/measure is referenced in table A.4 of part 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:

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

Dependability of Critical Computer Systems 2. F J Redmill, Elsevier Applied Science, 1989. ISBN 1-85166-381-9.

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

C.2.6 Design and coding standards

NOTE This technique/measure is referenced in table A.4 of part 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 and the related coding standards to be followed, for example JSP, MASCOT, Petri nets etc;

— the related coding standards to be followed (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. Software in the Safety Systems of Nuclear Power Stations, Annex A. International Electrotechnical Commission, 1987.

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

Verein Deutscher Ingenieure. Software-Zuverlassigkeit - Grundlagen, Konstruktive Massnahmen, 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 part 3.

Aim: To facilitate verifiability of the produced code.

Description: The detailed rules to be adhered to are fully agreed before coding. These rules comprise typically:

— details of modularisation, for example interface shapes, software module sizes;

— use of encapsulation, inheritance (restricted in depth) and polymorphy, in the case of object oriented languages;

— limited use or avoidance of certain language constructs, for example “goto”, “equivalence”, dynamic objects, dynamic data, dynamic data structures, recursion, pointers, exits etc; and

— restrictions on the permission of interrupts during safety-critical parts;

— layout of the code (listing).

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

C.2.6.3 No dynamic variables or dynamic objects

NOTE This technique/measure is referenced in table B.1 of part 3.

Aim: To exclude:

— unwanted or undetected overlay of memory;

— bottlenecks of resources during (safety related) runtime.

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 runtime. 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.

C.2.6.4 Online checking during creation of dynamic variables

NOTE This technique/measure is referenced in table B.1 of part 3.

Aim: To check that the memory to be allocated to a dynamic variable is free before allocation takes place, ensuring that the allocation of dynamic variables during runtime 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 runtime (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 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 has been used (for example, after exiting a subroutine) the whole memory which was allocated to it must be freed.

C.2.6.5 Limited use of interrupts

NOTE This technique/measure is referenced in table B.1 of part 3.

Aim: To keep software verifiable and testable.

Description: The use of interrupts shall be restricted. Interrupts may be used if they simplify the system. Software handling of interrupts must 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 shall be thoroughly documented.

C.2.6.6 Limited use of pointers

NOTE This technique/measure is referenced in table B.1 of part 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 part 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 part 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 statically 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:

Notes on structured programming. E W Dijkstra, Structured Programming, Academic Press, London, 1972, ISBN 0-12-200550-3.

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

Assessing a Class of Software Tools. M A Hennell et al. 7th Int. Conf. Software Engineering, Orlando, March 1984.

A Software Tool for Top-down Programming. D C Ince. Software - Practice and Experience, Vol 13, No 8, August 1983.

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

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

C.2.8 Information hiding/encapsulation

NOTE This technique/measure is referenced in table B.9 of part 3.

Aim: To increase the reliability and maintainability of software.

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

Information hiding is a general approach for minimising these difficulties. The key data structures are ‘hidden’ and can only be manipulated through a defined set of access procedures. This allows the internal structures to be modified or further procedures to be added without affecting the functional behaviour of the remaining software. For example, a name directory might have access procedures insert, delete and find. The access procedures and internal data structures could be re-written (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.

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

References:

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

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

C.2.9 Modular approach

NOTE This technique/measure is referenced in tables A.4 and B.9 of part 3.

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

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

— a software module 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 2 to 4 screen sizes;

— subprograms should have a single entry and a single exit only;

— software modules shall 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;

— each software module should hide something from its environment;

— any software modules interface should contain the minimum number of parameters necessary for its function; and

— a suitable restriction of parameter number should be specified, typically 5.

Reference: Structured Design - Fundamentals of a Discipline of Computer Program and Systems Design. E Yourdon, L L Constantine, Prentice-Hall, ISBN 0-13-854471-9, 1979.

C.2.10 Use of trusted/verified software modules and components

NOTE 1 This technique/measure is referenced in table A.4 of part 3.

NOTE 2 See also annex D.

Aim: To avoid the need for software modules and hardware component designs 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.

Description: This measure verifies that the software modules and components are sufficiently free from systematic design faults and/or operational failures. Only in rare cases will the employment of trusted software modules and components (ie those which are proven in use) be sufficient as the sole measure to ensure that the necessary safety integrity is achieved. For complex components with many possible functions (for example an operating system) it is essential to establish which functions are actually sufficiently proven in use. For example, where a self test routine is provided to detect hardware faults, if no hardware failure occurs within the operating period, one cannot consider the self test routine for fault detection as being proven by use.

A component or software module can be sufficiently trusted if it is already verified to the required safety integrity level, or 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 year with a confidence of 95% requires 300 years of operating experience,

  — 10^{-5} per year with a confidence of 99.9% requires 690 000 years of operating experience;

— all of the operating experience must relate to a known demand profile of the software module's functions, to ensure that increased operating experience genuinely leads to an increased knowledge of the software module's behaviour relative to that demand profile;

— no safety related failures.

To enable verification that a component or software module fulfils the criteria, the following must be documented:

- exact identification of each systems and its components, 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.

Reference: DIN V VDE 0801 A1: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben (Principles for Computers in Safety-Related Systems). Änderung 1 zu DIN V VDE 0801/01.90. Beuth-Verlag, Berlin, 1994.

C.3 Architecture design

C.3.1 Fault detection and diagnosis

NOTE This technique/measure is referenced in table A.2 of part 3.

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

Description: Fault detection is the 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 components, 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) and diversity (software faults). Some sort of voting is needed to decide on the correctness of results. Special methods applicable are: assertion programming, N-version programming and the safety bag technique; and 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 part 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.

References:

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

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

C.3.3 Failure assertion programming

NOTE This technique/measure is referenced in table A.2 of part 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 stops with the error.

For example,

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

References:

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

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

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

C.3.4 Safety bag

NOTE This technique/measure is referenced in table A.2 of part 3.

Aim: To protect against residual specification and implementation faults in software which adversely affect safety.

Description: A safety bag is an external monitor, implemented on an independent computer to a different specification. This safety bag is solely concerned with ensuring that the main computer performs safe, not necessarily correct, actions. The safety bag continuously monitors the main computer. The safety bag prevents the system from entering an unsafe state. In addition, if it detects that the main computer is entering a potentially hazardous state, the system has to be brought back to a safe state either by the safety bag or the main computer.

Reference: Using AI Techniques to Improve Software Safety. Proc. IFAC SAFECOMP 88, Sarlat, France, Pergamon Press, Oct 1986.

C.3.5 Software diversity (diverse programming)

NOTE This technique/measure is referenced in table A.2 of part 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.

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 but it provides a measure to detect and mask before they can affect safety.

References:

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

A Theoretical Basis for the Analysis of Multi-version Software subject to Co-incident Failures. D E Eckhardt and L D Lee, IEEE Trans SE-11 (12), 1985.

Computers can now perform vital safety functions safely. Otto Berg von Linde, Railway Gazette International, Vol 135, No 11, 1979.

C.3.6 Recovery block

NOTE This technique/measure is referenced in table A.2 of part 3.

Aim: To increase the likelihood of the program performing its intended function.

Description: Several different program sections are written, often independently, each of which is intended to perform the same desired function. The final program is constructed from these sections. The first section, called the primary, is executed first. This is followed by an acceptance test of the result it calculates. If the test is passed then the result is accepted and passed on to subsequent parts of the system. If it fails, any side effects of the first are reset and the second section, called the first alternative, is executed. This too is followed by an acceptance test and is treated as in the first case. A second, third or even more alternatives can be provided if desired.

References:

System Structure for Software Fault Tolerance. B Randall. IEEE Trans Software Engineering, Vol SE-1, No 2, 1975.

Fault Tolerance - Principles and Practice. T Anderson, P A Lee, Prentice Hall, 1981.

C.3.7 Backward recovery

NOTE This technique/measure is referenced in table A.2 of part 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.

C.3.8 Forward recovery

NOTE This technique/measure is referenced in table A.2 of part 3.

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

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

C.3.9 Re-try fault recovery mechanisms

NOTE This technique/measure is referenced in table A.2 of part 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: The Theory and Practice of Reliable System Design. D P Siewiorek and R S Schwarz. Digital Press.

C.3.10 Memorising executed cases

NOTE This technique/measure is referenced in table A.2 of part 3.

Aim: To force the software to fail safely if it attempts to execute a path which is not allowed.

Description: All relevant details of each program execution is documented. During normal operation each program execution is compared with the previously documented details. If it differs, a safety action is taken.

The execution documentation can contain the sequence of the individual decision-to-decision paths (DD paths) or the sequence of the individual accesses to arrays, records or volumes, or both.

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

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

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

C.3.11 Graceful degradation

NOTE This technique/measure is referenced in table A.2 of part 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. It must be taken into account from the top-most design phase.

References:

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

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

Fault Tolerance, Principle and Practices. T Anderson and P A Lee, Vol 3 of Dependable Computing and Fault-Tolerant Systems, Springer Verlag, ISBN 3-211-82077-9, 1987.

C.3.12 Artificial intelligence fault correction

NOTE This technique/measure is referenced in table A.2 of part 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 such that faults may be corrected and the effects of failures be minimised, in order to meet the desired safety integrity.

References:

Automatic Programming Techniques Applied to Software Development: An approach based on exception handling. M Bidoit et al, Proc. 1st Int. Conf. on Applications of Artificial Intelligence to Engineering Problems, Southampton, 165-177, 1986.

Artificial Intelligence and the Design of Expert Systems. G F Luger and W A Stubblefield, Benjamin/Cummings, 1989.

C.3.13 Dynamic reconfiguration

NOTE This technique/measure is referenced in table A.2 of part 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.

References:

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

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

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 part 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 types, but strict checks are imposed to ensure the correct type is used. These checks are imposed over the whole program, even if this is built from separately compiled units. The checks also ensure that the number and the type of procedure arguments match even when referenced from separately compiled software modules.

Strongly typed languages also 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.

References:

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-815A, 1983.

In Search of Effective Diversity: a Six Language Study of Fault-Tolerant Flight Control Software. A Avizienis, M R Lyu and W Schutz. 18th Symposium on Fault-Tolerant Computing, Tokyo, Japan, 27-30 June 1988, IEEE Computer Society Press, ISBN 0-8186-0867-6, 1988.

C.4.2 Language subsets

NOTE This technique/measure is referenced in table A.3 of part 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:

Requirements for programming languages in safety and security software standard. B A Wichmann. Computer Standards and Interfaces. Vol 14 pp 433-441, 1992.

Safer C: Developing Software for High-integrity and Safety-critical Systems. L Hatton, McGraw-Hill, ISBN 0-07-707640-0, 1994.

C.4.3 Certified tools and certified translators

NOTE This technique/measure is referenced in table A.3 of part 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.

References:

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

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

C.4.4 Translator: increased confidence from use

NOTE This technique/measure is referenced in table A.3 of part 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, whose correct performance has been demonstrated in many projects already. Translators without operating experience or with any serious known faults are prohibited.

If the translator has shown small deficiencies, the related language constructs are noted down and carefully avoided during a safety related project.

Another version to this way of working is to restrict the usage of the language to only its commonly used features.

This recommendation is based on the experience from many projects. It has been shown that immature translators are a serious handicap to any software development. They make a safety-related software development generally infeasible.

It is also known, presently, that no method exists to prove the correctness for all compiler parts.

C.4.5 Library of trusted/verified software modules and components

NOTE This technique/measure is referenced in table A.3 of part 3.

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

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

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

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

C.2.10 describes one approach by which a software module may be classified as trusted.

References:

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

DIN V VDE 0801 A1: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben (Principles for Computers in Safety-Related Systems). Änderung 1 zu DIN V VDE 0801/01.90. Beuth-Verlag, Berlin, 1994.

C.4.6 Suitable programming languages

NOTE This technique/measure is referenced in table A.3 of part 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 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 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 sub-ranges; 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; and

— dynamic resource management (with run time checking).

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 machine orientated nature.

Table C.1 gives recommendations for specific programming languages.

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	MODULA-2	HR	HR	R	R
4	MODULA -2 with subset	HR	HR	HR	HR
5	PASCAL	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	PL/M	R	---	NR	NR
12	PLM with subset and coding standard	HR	R	R	R
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
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 recommendations R, HR and -- are explained in annex A of part 3.

NOTE 2 System software includes the operating system, drivers embedded oriented 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. Wherever possible tests 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. Wherever possible tests 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.

References:

IEC 60880. Software in the Safety Systems of Nuclear Power Stations, Annex A. International Electrotechnical Commission, 1987.

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

C.5 Verification and modification

C.5.1 Probabilistic testing

NOTE This technique/measure is referenced in tables A.5, A.7 and A.9 of part 3.

Aim: To gain a quantitative figure about the reliability properties of the investigated software.

Description: 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 tests 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 view points. 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:

Software Testing via Environmental Simulation (CONTESSE Report). Available until December 1998 from: Ray Browne, CIID, DTI, 151 Buckingham Palace Road, London, SW1W 9SS, UK, 1994.

Validation of ultra high dependability for software based systems. Littlewood and Strigini. Comm. ACN, 36 (11), 69-80, 1993.

Handbook of Software Reliability Engineering. M R Lyu (Ed). IEEE Computer Society Press, McGraw Hill, 1995. ISBN 0-07-039400-8.

C.5.2 Data recording and analysis

NOTE This technique/measure is referenced in tables A.5 and A.8 of part 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 1-85166-381-9.

C.5.3 Interface testing

NOTE This technique/measure is referenced in table A.5 of part 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 part 3.

Aim: To remove software errors occurring at parameter limits or boundaries.

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

— zero divisor;

— blank ASCII characters;

— empty stack or list element;

— 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, 1 and 2 elements.

References:

IEC 61704: Guide to test methods for dependability assessment of software, June 1995.

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

C.5.5 Error guessing

NOTE This technique/measure is referenced in tables B.2 and B.8 of part 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. G Myers, Wiley & Sons, New York, USA, 1979.

C.5.6 Error seeding

NOTE This technique/measure is referenced in table B.2 of part 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.

$$\frac{\textit{Found seeded errors}}{\textit{Total number of seeded errors}} = \frac{\textit{Found real errors}}{\textit{Total number of real errors}}$$

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.

C.5.7 Equivalence classes and input partition testing

NOTE This technique/measure is referenced in tables B.2 and B.3 of part 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:

IEC 61704: Guide to test methods for dependability assessment of software, June 1995.

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

C.5.8 Structure based testing

NOTE This technique/measure is referenced in table B.2 of part 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 such 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.

— **Statements**: this is the least rigorous test since it is possible to execute all code statements without exercising both branches of a conditional statement.

— **Branches**: both sides of every branch should be checked. This may be impractical for some types of defensive code.

— **Compound conditions**: every condition in a compound conditional branch (ie linked by AND/OR) is exercised.

— **LCSAJ**: a linear code sequence and jump is any linear sequence of code statements, including conditional statements, terminated by a jump. Many potential sub-paths will be infeasible due to constraints on the input data imposed by the execution of earlier code.

— **Data flow**: the execution paths are selected on the basis of data usage; for example a path where the same variable is both written and read.

— **Call graph**: a program is composed of subroutines which may be invoked from other subroutines. The call graph is the tree of subroutine invocations in the program. Tests are designed to cover all invocations in the tree.

— **Entire path**: execute all possible paths through the code. Complete path testing is almost always infeasible due to the exceedingly large number of potential paths.

References:

IEC 61704: Guide to test methods for dependability assessment of software, June 1995.

Reliability of the Path Analysis Testing Strategy. W Howden. IEEE Trans Software Engineering, Vol SE-3, 1976.

C.5.9 Control flow analysis

NOTE This technique/measure is referenced in table B.8 of part 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, which is well structured code whose control graph is reducible by successive graph reductions to a single node. poorly structured code can only be reduced to a knot composed of several nodes.

References:

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

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

C.5.10 Data flow analysis

NOTE This technique/measure is referenced in table B.8 of part 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:

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

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

C.5.11 Sneak circuit analysis

NOTE This technique/measure is referenced in table B.8 of part 3.

Aim: To detect an unexpected path or logic flow within a system which, under certain conditions, initiates an undesired function or inhibits a desired function.

Description: A sneak circuit path may consist of hardware, software, operator actions, or combinations of these elements. Sneak circuits are not the result of hardware failure but are latent conditions inadvertently designed into the system or coded into the software programs, which can cause it to malfunction under certain conditions.

Categories of sneak circuits are:

— sneak paths which cause current, energy, or logical sequence to flow along an unexpected path or in an unintended direction;

— sneak timing in which events occur in an unexpected or conflicting sequence;

— sneak indications which cause an ambiguous or false display of system operating conditions, and thus may result in an undesired action by the operator;

— sneak labels which incorrectly or imprecisely label system functions, for example, system inputs, controls, displays, buses, etc, and thus may mislead an operator into applying an incorrect stimulus to the system;

Sneak circuit analysis relies on the recognition of basic topological patterns with the hardware or software structure (for example six basic patterns have been proposed for software). Analysis takes place with the aid of a checklist of questions about the use and relationships between the basic topological components.

References:

Sneak Analysis and Software Sneak Analysis. S G Godoy and G J Engels. J. Aircraft Vol 15, No 8, 1978.

Sneak Circuit Analysis. J P Rankin, Nuclear Safety, Vol 14, No 5, 1973.

C.5.12 Symbolic execution

NOTE This technique/measure is referenced in table B.8 of part 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 can be checked against the expected expression.

References:

Formal Program Verification using Symbolic Execution. R B Dannenberg and G W Ernst. IEEE Transactions on Software Engineering, Vol SE-8, No 1, 1982.

Symbolic Execution and Software Testing. J C King, Communications of the ACM, Vol 19, No 7, 1976.

C.5.13 Formal proof

NOTE This technique/measure is referenced in table A.9 of part 3.

Aim: To prove the correctness of a program without executing it, using theoretical and mathematical models and rules.

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

Several formal methods are described in this overview, for instance, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z (See C.2.4 for descriptions of these methods).

Reference: Can Program Proving be made Practical. J Dahl, Research Report, ISBN 82-90230-26-5 No 33, Oslo, May.

C.5.14 Complexity metrics

NOTE This technique/measure is referenced in tables A.9 and A.10 of part 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.

References:

Software Metrics: A Rigorous and Practical Approach. N E Fenton, International Thomson Computer Press, ISBN 1-85032-275-9, 2nd Edition, 1996.

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

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

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

C.5.15 Fagan inspections

NOTE This technique/measure is referenced in table B.8 of part 3.

Aim: To reveal mistakes and faults in all phases of the program development.

Description: A ‘formal’ audit on quality assurance documents aimed at finding mistakes and faults. The inspection procedure consists of five stages: planning, preparation, inspection, rework and follow up. Each of these stages has its own separate objective. The complete system development (specification, design, coding and testing) must be inspected.

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

C.5.16 Walkthroughs/design reviews

NOTE This technique/measure is referenced in table B.8 of part 3.

Aim: To detect faults in some product of the development as soon and as economically as possible.

Description: IEC has published a guide on formal design reviews, which includes a general description of formal design reviews, their objectives, details of the various design review types, the composition of a design review team and their associated duties and responsibilities. The IEC document also provides general guidelines for planning and conducting formal design reviews, as well as specific details concerning the role of independent specialists within a design review team. Examples of specialist functions include, amongst others, reliability, maintenance support and availability.

The IEC recommend that a “formal design review should be conducted for all new products/processes, new applications, and revisions to existing products and manufacturing processes which affect the function, performance, safety, reliability, ability to inspect maintainability, availability, ability to cost, and other characteristics affecting the end product/process, users or bystanders”.

A code walkthrough consists of a walkthrough team selecting a small set of paper test cases, representative sets of inputs and corresponding expected outputs for the program. The test data is then manually traced through the logic of the program.

References:

IEC 61160: Formal Design Review. 1st Edition (Amendment 1), 1994.

Software Inspection. T Gilb, D Graham, Addison-Wesley, ISBN 0-201-63181-4, 1993.

C.5.17 Prototyping/animation

NOTE This technique/measure is referenced in tables B.3 and B.5 of part 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 sub-set of system functions, constraints, and performance requirements are selected. A prototype is built using high level tools. At this stage, constraints such as the target computer, implementation language, program size, maintainability, reliability and availability need not be considered. The prototype is evaluated against the customer's criteria and the system requirements may be modified in the light of this evaluation.

References:

The emergence of rapid prototyping as a real-time software development tool. J E Cooling, T S Hughes, Proc. 2nd Int. Conf. on Software Engineering for Real-time Systems, Cirencester, UK, IEE, 1989.

Software evolution through rapid prototyping. Luqi, IEEE Computer, 22 (5), 13-27, May 1989.

Approaches to Prototyping. R Budde et al, Springer Verlag, ISBN 3-540-13490-5, 1984.

Proc. Working Conference on Prototyping. Namur Oct 1983, Budde et al, Springer Verlag, 1984.

Using an executable specification language for an information system. S Urban et al. IEEE Trans Software Engineering, Vol SE-11 No 7, July 1985.

C.5.18 Process simulation

NOTE This technique/measure is referenced in table B.3 of part 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:

Software Testing via Environmental Simulation (CONTESSE Report). Available until December 1998 from: Ray Browne, CIID, DTI, 151 Buckingham Palace Road, London, SW1W 9SS, UK, 1994.

A Software Simulator - An Aid to Plant Commissioning. S Nunns, EWICS TC7.

Physical Fault Simulation. F Morillon, EWICS Doc No WP460.

C.5.19 Performance requirements

NOTE This technique/measure is referenced in table B.6 of part 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.

C.5.20 Performance modelling

NOTE This technique/measure is referenced in tables B.2 and B.5 of part 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 is suitable 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: The Design of Real-time Systems: From Specification to Implementation and Verification. H Kopetz et al, Software Engineering Journal, 72-82, 1991.

C.5.21 Avalanche/stress testing

NOTE This technique/measure is referenced in table B.6 of part 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.

C.5.22 Response timing and memory constraints

NOTE This technique/measure is referenced in table B.6 of part 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 part 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. The possible decisions are:

— only the changed software module is reverified;

— all affected software modules are reverified; or

— the complete system is reverified.

Reference: Dependability of Critical Computer Systems 2. F J Redmill, Elsevier Applied Science, 1989. ISBN 1-85166-381-9.

C.5.24 Software configuration management

NOTE This technique/measure is referenced in table A.8 of part 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. 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 components). In particular, systems or subsystems can be reliably re-built from consistent sets of component versions.

References:

Configuration Management Practices for Systems, Equipment, Munitions and Computer Programs. MIL-STD-483.

Software Configuration Management. J K Buckle. Macmillan Press, 1982.

Software Configuration Management. W A Babich. Addison-Wesley, 1986.

Configuration Management Requirements for Defence Equipment. UK Ministry of Defence Standard 05-57 Issue 1, 1980.

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 part 3.

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

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

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

Both methods are potentially executable if used as specifications.

C.6.2 Hazard and Operability Study (HAZOP)

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.

At a later time, it is often necessary to carry out further hazard analysis (often referred to as probabilistic or quantitative risk assessment), to consider the major hazards in more detail.

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. It is helpful if: a fixed time schedule is allocated within the project for the meetings; each one is scheduled for at least half a day; and no more than four per week are scheduled, so that the flow of accompanying documentation is maintained. Documentation from the meetings will form a substantial part of the system hazard/safety dossier.

The HAZOP technique evolved in the process industry and is difficult to apply without modification to the software element of PES. Different derivative methods for PES HAZOPs (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:

Draft Interim Defence Standard 00-58/1: “A Guide to HAZOP Studies on Systems which Incorporate a Programmable Electronic System”. Ministry of Defence (UK). March 1995.

Hazard and Operability (HAZOP) studies applied to computer-controlled process plants. P Chung and E Broomfield. In “Computer Control and Human Error” by T Kletz, Institution of Chemical Engineers, 165-189 Railway Terrace, Rugby, CV1 3HQ, UK, ISBN 0-85295-362-3, 1995.

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

HAZOP and HAZAN. T A Kletz. Institution of Chemical Engineers, 165-189 Railway Terrace, Rugby, CV1 3HQ, UK, 3rd Edition, ISBN 0-85295-285-6, 1992.

A Guide to HAZOPS. Chemical Industries Association Ltd, 1977.

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

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

C.6.3 Common cause failure analysis

NOTE 1 This technique/measure is referenced in table A.10 of part 3.

NOTE 2 See also annex D of part 6.

Aim: To determine potential failures in multiple systems or multiple sub-systems 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:

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

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

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

C.6.4 Markov models

NOTE See B.1 of part 6 for a brief comparison of this technique against reliability block diagrams, in the context of analysing hardware safety integrity.

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

Description: A graph of the system is constructed. The graph represents the status of the system with regard to its failure states (the failure states are represented by the nodes of the graph). The edges between nodes, which represent the failure events or repair events, are weighted with the corresponding failure rates or repair rates. It is assumed that a change of state, N, to a subsequent state, N+1, is independent of the previous state, N-1. Note that 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.

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 life-cycle of the system since no simple combinatorial formulae exist for calculating the corresponding probabilities.

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

References:

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

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

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

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

C.6.5 Reliability block diagrams

NOTE This technique/measure is referenced in table A.10 of part 3 and is used in annex B of part 6.

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 61078: Analysis techniques for dependability - Reliability block diagram method, 1991.

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

Fault Tree Handbook. W E Vesely et al, NUREG-0942, Division of System Safety Office at Nuclear Reactor Regulation, US Nuclear Regulatory Commission, Washington, DC 20555, 1981.

C.6.6 Monte-Carlo simulation

NOTE This technique/measure is referenced in tables A.10 and B.4 of part 3.

Aim: To simulate real world phenomena in software using random numbers.

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.

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

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

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

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

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 components, 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: 1987, IEEE guide for general principles of reliability analysis of nuclear power generating station safety systems. An equivalent term, significance level, is used in IEC 61164: 1995, Reliability growth – Statistical test and estimation methods.

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 part 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 (ie 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 $<10^{-4}$	4.6×10^5	3×10^5	$\geq 10^{-9}$ to $<10^{-8}$	4.6×10^9	3×10^9
3	$\geq 10^{-4}$ to $<10^{-3}$	4.6×10^4	3×10^4	$\geq 10^{-8}$ to $<10^{-7}$	4.6×10^8	3×10^8
2	$\geq 10^{-3}$ to $<10^{-2}$	4.6×10^3	3×10^3	$\geq 10^{-7}$ to $<10^{-6}$	4.6×10^7	3×10^7
1	$\geq 10^{-2}$ to $<10^{-1}$	4.6×10^2	3×10^2	$\geq 10^{-6}$ to $<10^{-5}$	4.6×10^6	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-α, is given by:

$$p \le 1 - \sqrt[n]{a} \qquad \text{or} \qquad n \ge -\frac{\ln a}{p}$$

D.2.1.3 Example

Table D.2 — Probabilities of failure for low demand mode of operation

1-a	**p**
0.95	3/n
0.99	4.6/n

For a probability of failure on demand of SIL3 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 shutdown) that is being tested.

Table D.3 — Mean distances of 2 test points

Dimension of the domain	Mean distance of 2 test points in direction of an arbitrary axis
1	$d = 1/n$
2	$d = \sqrt[2]{1/n}$
3	$d = \sqrt[3]{1/n}$
k	$d = \sqrt[k]{1/n}$

NOTE k can be any positive integer. The values 1, 2 and 3 are just examples.

D.2.2.3 Example

Consider a safety shutdown 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:

$$l = -\frac{\ln a}{t}$$

The probability of failure is indirectly proportional to the mean time between failures:

$$l = \frac{1}{MTBF}$$

NOTE This standard does not distinguish between the probability of failure per hour and the rate of failures in an hour. Strictly, the probability of failure, F, is related to the failure rate, f, by the equation $F = 1 - e^{-ft}$, 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 - a	l
0.95	3/t
0.99	4.6/t

To verify that the mean time between failures is at least 10^8 hours with a confidence level of 95%, a test time of 3 x 10^8 hours 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 $C_{j,N} = \dfrac{N(N-1)\ldots(N-j+1)}{j!}$

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 x 10^6 runs have been executed. It is estimated that each 100th run fulfils the above prerequisites. So 7.5 x 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 having not tested everything equals 2.87 x 10^{-5}.

For N = 4000 the values of the first terms depending on n are:

Table D.5 — Probability of testing all program properties

n	p
5×10^{4}	$1 - 1.49 \times 10^{-2} + 1.10 \times 10^{-4} - ...$
7.5×10^{4}	$1 - 2.87 \times 10^{-5} + 4 \times 10^{-10} - ...$
1×10^{5}	$1 - 5.54 \times 10^{-8} + 1.52 \times 10^{-15} - ...$
2×10^{5}	$1 - 7.67 \times 10^{-19} + 2.9 \times 10^{-37} - ...$

D.3 References

Further information on the above techniques can be found in:

a) Verification and Validation of Real-Time Software, Chapter 5. W J Quirk (ed). Springer Verlag, 1985, ISBN 3-540-15102-8.

b) Combining Probabilistic and Deterministic Verification Efforts. W D Ehrenberger, SAFECOMP 92, ISBN 0-08-041893-7.

c) Ingenieurstatistik. Heinhold/Gaede, Oldenbourg, 1972, ISBN 3-486-31743-1.

Index