**Ministry of Defence**

# Defence Standard

## 00-55(PART 2)/Issue 2 1 August 1997

# REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT

# PART 2: GUIDANCE

**This Part 2 of Def Stan 00-55 supersedes INTERIM Def Stan 00-55/Issue 1 dated 5 April 1991**

## AMENDMENTS ISSUED SINCE PUBLICATION

| AMD NO | DATE OF ISSUE | TEXT AFFECTED | SIGNATURE & DATE |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

## Revision Note

Issue 2 of this Standard has been prepared to take account of comments received from users of Interim Issue 1 and to comply with current MOD policy.

## Historical Record

Interim Issue 1 Initial Publication 5 April 1991

# REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT

## PART 2: GUIDANCE

## PREFACE

This Part 2 of Def Stan 00-55
Supersedes INTERIM Def Stan 00-55 (Part 2)

i This Part of the Standard contains guidance on the requirements contained in Part 1. This guidance serves two functions: it elaborates on the requirements in order to make conformance easier to achieve and assess; and it provides technical background.

ii This Standard is one of a family of standards dealing with safety that is being developed or adopted by MOD, taking into account international standardization activities and supporting research and development.

iii This Standard has been agreed by the authorities concerned with its use and is intended to be used whenever relevant in all future designs, contracts, orders etc and whenever practicable by amendment to those already in existence. If any difficulty arises which prevents application of this Defence Standard, the Directorate of Standardization shall be informed so that a remedy may be sought.

iv Any enquiries regarding this Standard in relation to an invitation to tender or a contract in which it is incorporated are to be addressed to the responsible technical or supervising authority named in the invitation to tender or contract.

v This Standard has been devised for the use of the Crown and its contractors in the execution of contracts for the Crown. The Crown hereby excludes all liability (other than liability for death or personal injury) whatsoever and howsoever arising (including, but without limitation, negligence on the part of the Crown its servants or agents) for any loss or damage however caused where the Standard is used for any other purpose.

CONTENTS PAGE

REQUIREMENTS FOR SAFETY RELATED SOFTWARE IN DEFENCE EQUIPMENT

PART 2: GUIDANCE

## Section One. General

### 0 Introduction

This Part of the Defence Standard provides guidance on the requirements in Part 1. From section two onwards, this guidance is organized by the main clause and subclause headings used in Part 1; however, subsubclauses do not necessarily correspond exactly between the two Parts.

### 1 Scope

**1.1** This Part of the Standard provides information and guidance on the procedures necessary for the production of software of all levels of safety integrity. However, it places particular emphasis on describing the procedures necessary for specification, design, coding, production and in-service maintenance and modification of Safety Critical Software (SCS).

**1.2** It should be emphasized that safety is a system property and achieving and maintaining safety requires attention to all aspects of the system, including its human, electronic and mechanical components. This Standard addresses only one important component - ie the development of software to meet a predetermined safety integrity level. The achievement of safety targets by overall design, and in particular whether safety features are to be controlled by hardware, software or manual procedures, is not addressed. A systems approach to hazard analysis and safety risk assessment is explained in Def Stan 00-56.

**1.3** Where safety is dependent on the safety related software (SRS) fully meeting its requirements, demonstrating safety is equivalent to demonstrating correctness with respect to the Software Requirement. In other cases, safety may be dependent on the SRS behaving in accordance with an identifiable set of safety requirements, contained within the Software Requirement, rather than correctness with the total Software Requirement to provide the required safety integrity level. Because of the difficulties of separating safety properties from the other behavioural properties of the SRS and the need to demonstrate adequate partitioning between these properties, this Standard tends towards the former approach and assumes that correctness is equivalent to safety. However, providing that safety can be achieved and demonstrated, overall correctness need not be an objective from a safety point of view.

### 2 **WARNING**

This Standard refers to a number of procedures, techniques, practices and tools which when followed or used correctly will reduce but not necessarily eliminate the probability that the software will contain errors. This Standard refers only to technical suitability and in no way absolves either the designer, the producer, the supplier or the user from statutory and all other legal obligations relating to health and safety at any stage.

### 3 Related Documents

The related documents identified in Part 1 also apply to this Part of the Standard.

## **4** Definitions

The definitions in Part 1 apply also to this Part of the Standard.

## Section Two. Safety Management

### 5 Safety Management Activities

No guidance.

### 6 Software Safety Plan

**6.1** No guidance.

**6.2** No guidance.

**6.3** The contents of the Software Safety Plan should be as detailed in **annex B.1**. On some projects with a particularly long timescales it may not be possible to specify CVs of key individuals nor specify the qualifications required of all the staff who might be involved with the production of the SRS. This could be considered when agreeing the Software Safety Plan with MOD.

**6.4** No guidance.

### 7 Software Safety Case

#### 7.1 General

**7.1.1** A key product of the SRS development is the Software Safety Case. The Software Safety Case is a readable justification that the SRS is safe in its specified context; ie that it meets its requirements insofar as they impact on safety. In an MOD context, it should facilitate the acceptance of the SRS by the MOD PM and any regulatory or safety bodies involved. The content of the Software Safety Case should be as detailed in **annex B.2**.

**7.1.2** In practice there will be several different audiences for the Software Safety Case during the project lifecycle, each with their own viewpoint, and these will require different levels of detail. The Software Safety Case should therefore be structured to allow it to be read at different levels of depth. The Software Safety Case should also contain system level information that assists the reader in understanding the context of the SRS and in judging the adequacy of the arguments presented in the Software Safety Case. Detailed supporting evidence should be recorded in the Software Safety Records Log.

**7.1.3** The Software Safety Case includes feedback from in-service performance because it is of great value in confirming or otherwise the original safety case and the effectiveness of the safety and software engineering techniques deployed, but depends upon accurate monitoring, recording and analysis. It can also provide important evidence for improving the SRS development process and demonstrating ALARP. This Standard therefore requires a DRACAS to be set up (see clause **42**).

**7.1.4** This Standard requires an appraisal of in-service experience with relevant variants of the equipment. It may be convenient to structure the Software Safety Case for the variants as one safety case with common chapters for those aspects common to the variants.

**7.1.5** The major sources of information used in constructing the Software Safety Case are shown in guidance **annex E.**

**7.1.6** **Figure 1** illustrates how the key documents concerning the safety issues are related to one another. Because this is a summary, only the top-level documents are included. Reference should be made to the relevant clauses of the Standard for more detailed descriptions and definitions.

**7.1.7** The Software Requirement and Hazard Log are system level documents that form inputs to the SRS development process. The Software Requirement contains the requirements which, if met by the SRS, ensure that the SRS will operate in a safe manner. The Hazard Log catalogues the hazards that have been identified from the operation of the system within which the SRS is to operate and, consequently, enables possible unsafe behaviours of the system to be associated with the way in which the SRS operates.

**7.1.8** The Software Safety Plan describes the means by which the safety requirements that are embodied in the Software Requirement and in the Hazard Log are to be discharged.

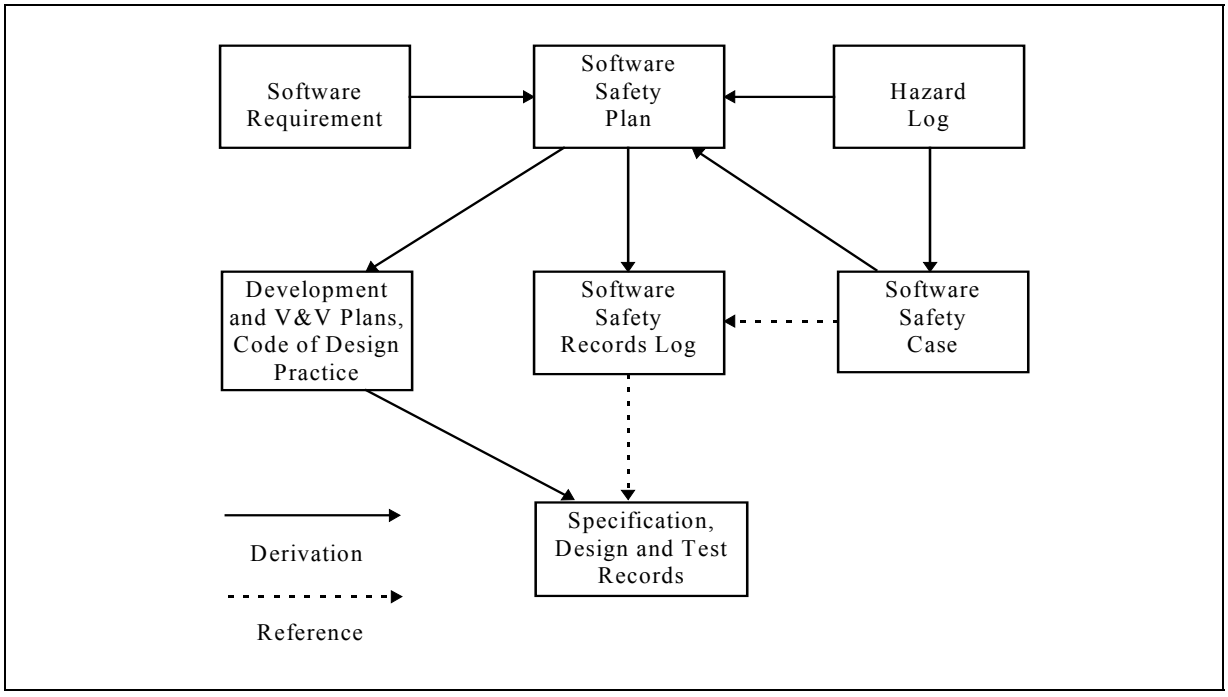



Figure 1: Top Level Document Structure

**7.2** Development of the Software Safety Case

**7.2.1** The Software Safety Case will form a part of the safety case for the system. This contains a justification that the requirements for the SRS include requirements which, if met, will ensure the safety of the SRS component of the system.

**7.2.2** The Software Safety Case evolves during SRS development to take account of the evidence that is accumulated and any changes to the Software Safety Plan.

**7.2.3** The Software Safety Case is a living document that is formally issued at defined project stages to the MOD PM. The development of the Software Safety Case will proceed through a number of stages of increasing detail as the project progresses. At the beginning of a project confidence is required, before committing significant resources, that there is minimal risk of failing to meet the safety requirements. During the project confidence is required that the safety requirements are being met; and during use confidence is required that the equipment is safe and continues to be safe as the operational requirements and the environment change.

**7.3** Safety arguments

**7.3.1** The justification of how the SRS development process is expected to deliver SRS of the required safety integrity level, mainly on the basis of the performance of the process on previous projects, is covered in **7.4** and **annex E**. However, in general the process used is a very weak predictor of the safety integrity level attained in a particular case, because of the variability from project to project. Instrumentation of the process to obtain repeatable data is difficult and enormously expensive, and capturing the important human factors aspects is still an active research area. Furthermore, even very high quality processes only predict the fault density of the software, and the problem of predicting safety integrity from fault density is insurmountable at the time of writing (unless it is possible to argue for zero faults).

**7.3.2** However, some aspects of the process, such as the construction of formal arguments and statistical validation testing, do provide a direct measure of the safety integrity level of the SRS. Part 1 of this Standard requires that the attainment of the target safety integrity level of the SRS is argued by means of these direct measures.

**7.3.3** There are two basic types of safety argument:

a) Analytical arguments, which are:

i) formal arguments that the object code satisfies the formal specification and that the formal specification complies with the safety properties in the Software Requirement;

ii) arguments based on analysing the performance of the constituent parts of the system and from these inferring the performance of the system as a whole.

b) Arguments based on observing the behaviour of the SRS.

**7.3.4** These arguments are diverse in the sense that they are based on different evidence and use different analysis methods with different underlying assumptions. Both types of argument are required by the Standard because of the coverage, fallibility and uncertainty associated with any single argument, especially when the Software Design Authority is attempting to justify very small probabilities of failure (for example one failure in 105 years ). Straightforward adherence to this Standard will enable analytical arguments to be made using formal methods, and observational arguments to be made from statistical validation testing.

**7.3.5** The Software Design Authority may in addition be able to employ other arguments, which may make it easier to justify the attainment of the highest safety integrity levels. The possible safety arguments are listed in table 1, with an indication of the way they scale with software size and safety integrity level, the assumptions on which they are based, their

**7.3.5** (Cont)

limitations, and the maximum safety integrity level for which they are practicable and are likely to give adequate assurance. Expanded information on safety arguments is given in **annex E**.

**7.4** Justification of the SRS development process

**7.4.1** The Software Design Authority should justify how the SRS development process is expected to deliver SRS of the required safety integrity. **Annex E** provides further guidance.

**7.4.2** A safety analysis of the SRS development process described in the Code of Design Practice and Software Safety Plan should be carried out, explaining how each technique will contribute to the assurance of the SRS at the applicable process stages. The analysis should include a description of the fault avoidance techniques to be applied, such as the use of a strongly-typed programming language to avoid certain programming faults. It should also address the independent checks that may be possible, and the possibility of human error in applying these checks.

**7.4.3** Historical evidence from previous projects and/or the published literature should be presented to show that highly dependable software has been successfully produced using such a development process. More details are provided in **annex E**.

The use of commercial off-the-shelf (COTS) or other previously developed software should be justified, and the way in which it contributes to the final safety integrity of the SRS explained.

Table 1 Safety Arguments

| **Argument** | **Scaling with size and safety integrity level (SIL)** | **Assumptions and limitations** | **Max SIL** |
|---|---|---|---|
| Formal arguments | About linear with code size. Limited complexity of application. Some resource related properties or concurrent aspects difficult to address.<br><br>Policy for formal proof vs. rigorous arguments needs careful justification. | Evidence very strong for properties amenable to this approach.<br><br>Very dependent on system design.<br><br>Validity of rigorous arguments for assurance (as opposed to development) hard to quantify. | 4 |
| Validation testing | Effort increases with complexity of equipment by an order of magnitude for each SIL. Highest SILs require multiple copies and/or accelerated testing.<br><br>Practicable up to SIL 4 for systems that respond to demands (eg shut-down systems);impracticable over SIL 1 for continuously operating systems. | Operational profile may be hard to justify.<br><br>Potentially high costs of checking results and significant elapsed time. Costs of repeating tests following minor changes can be significant. | 4 |
| Exhaustive testing | Not dependent on SIL but very sensitive to complexity of software. | Unlikely to be practicable except for special cases, which may be readily tractable by proof anyway | 4 |
| Fail safety | Independent of size or SIL in theory, but technique may require use of encryption and specialised hardware that makes it hard to scale up. | Very appealing for fail safe applications but usually only applied to certain fault classes, eg to compiler and hardware faults. | 4 |
| Experience in similar environment | The amount of experience needed increases with an order of magnitude for each SIL. Only viable normally for systems with multiple copies. | Limitations of reporting schemes, problems in justifying similarity of environment.<br><br>Normally only applicable for continuously operating systems or those with frequent demands placed on them.<br><br>May be appealing for mass produced components in environments that do not change (eg operating system firmware). | 2/3 |

## 8 Safety Analysis

### 8.1 General

**8.1.1** The safety of the SRS cannot be considered in isolation from the system as a whole, and safety analysis therefore has to begin at the system level. The requirements for safety analysis are given in detail in Def Stan 00-56, but in outline the process is as follows:

a) Determine the tolerable safety risk from the equipment.

b) Identify and analyse the potential accidents involving the equipment, and the associated hazards.

c) Design the equipment to remove or reduce the hazards so that the tolerable safety risk is not exceeded. This includes:

i) specifying any safety features that may be required;

ii) apportioning safety integrity levels to hardware and software components;

iii)defining the strategy for fault detection, fault tolerance and fail safety as appropriate, including any requirements for technological diversity or segregation.

**8.1.2** This process is carried out iteratively at increasing levels of detail as the equipment is developed, by the Design Authority on the equipment as a whole, and by the Software Design Authority on the SRS components.

**8.1.3** Safety analysis of the SRS includes establishing the correct safety integrity level for each software component. Safety integrity is a measure of the likelihood of dangerous failure, and therefore may differ from reliability, which refers to the likelihood of delivering the specified, operational service. Equipment may be safe but unreliable, for instance if fail safe actions keep occurring; or it may be reliable but unsafe, if the few failures that do occur are mostly dangerous.

**8.1.4** Safety integrity is defined as four levels, S1 to S4, where S1 is the least critical and S4 is safety critical. Def Stan 00-56 describes a process for defining these levels, and provides guidance on numerical values. Safety integrity levels are also addressed in IEC 1508, where they are defined in terms of the probability of dangerous failure on the basis of a judgement about the levels of safety integrity achieved in practice. These definitions are summarised in table 2, which provide the combined probability of failure from all causes, including, for example, hardware wear-out failures and programming errors.

**8.1.5** Under certain circumstances, the safety integrity level of a system may be reduced by means of technological diversity such as the use of a hardware component to provide essential safety functions. Def Stan 00-56 contains the rules for this.

Table 2 Failure Probability for Different Safety Integrity Levels

| **Safety integrity level** | **Safety integrity** (probability of dangerous failure per year, per system) |
|---|---|
| S4 | $\geq 10^{-5}$ to $< 10^{-4}$ |
| S3 | $\geq 10^{-4}$ to $< 10^{-3}$ |
| S2 | $\geq 10^{-3}$ to $< 10^{-2}$ |
| S1 | $\geq 10^{-2}$ to $< 10^{-1}$ |

**8.2** No guidance.

**8.3** No guidance.

## 9 Software Safety Records Log

The Software Development Plan, the Software Verification and Validation Plan and the Code of Design Practice between them determine the way in which the SRS is developed and verified. The methods, procedures and practices have an impact on the integrity of the SRS and these documents will therefore be required by the Software Safety Plan and referenced by the Software Safety Case. Detailed evidence that the plans and practices have been followed are accumulated in such forms as design records, records of reviews and results of V&V. These are referenced by the Software Safety Records Log and contribute to the Software Safety Case. Guidance on the supporting analyses is provided in **annex E**.

The contents of the Software Safety Records Log should be as detailed in **annex B.3**.

## 10 Software Safety Reviews

**10.1** The software safety reviews should be carried out at intervals during the SRS development process. During phases of the project in which safety issues have particular prominence and greatest impact upon the development programme (typically, the earlier phases) the frequency of software safety reviews is likely to be particularly high. As an absolute minimum, there should be a software safety review at the end of each phase. In addition to those reviews that are scheduled as a matter of course in the Software Safety Plan, a software safety review should be convened in the event that safety concerns arise during the course of the SRS development. Software Safety Reviews may be included as part of the normal software review programme and the system safety programme.

**10.2** The participants in a software safety review should be all those who can contribute relevant information or have an interest in the safety of the SRS. Typically, they will include, as appropriate:

a) the Software Project Safety Engineer (as chairman);

**10.2** (Cont)

b) the individual from the Software Design Authority responsible for safety certification (if different);

c) the Project Safety Engineer (if different);

d) the Software Project Manager (or representative);

e) members of the Design Team;

f) members of the V&V Team;

g) the Independent Safety Auditor (by invitation); h) the MOD PM (by invitation);

i) representatives of the system users (by invitation).

**10.3** The objective of each software safety review should be to review the evidence that contributes to the Software Safety Case in accordance with the Software Safety Plan. Software safety reviews should include checks of traceability reviews, especially concerning any requirements that are particularly related to safety. Any failures during the SRS development that might indicate a possible shortfall in the Software Safety Plan or in its execution should be a particular cause of concern.

**10.4** The software safety review should agree any actions that need to be taken in order to ensure the continued safety integrity of the SRS and its development. The Software Project Safety Engineer is responsible for ensuring that these actions are carried out but in many cases will require the support of the Software Project Manager who should therefore be represented at the review meetings.

**10.5** No guidance.

**10.6** No guidance.

**11** Software Safety Audits

**11.1** Software safety audits provide an independent assessment of the safety aspects of the SRS and of its development. The Independent Safety Auditor should take the role of a customer of the Software Safety Case and the auditing activities should be aimed at establishing the adequacy and correctness of the arguments made in the Software Safety Case by analysis of the arguments used, by testing of the evidence presented in supporting documentation such as the Software Safety Plan and the Software Safety Records Log and by auditing the project against the standards and procedures being followed.

**11.2** The Software Safety Audit Plan should be updated at least at the start of the specification, design, coding and testing processes. The contents of the Software Safety Audit Plan should be as detailed in **annex B.4**.

**11.3** The contents of the Software Safety Audit Report should be as detailed in **annex B.5**. Each Software Safety Audit Report should be available to those whose activities have been audited. If, as is likely, a number of Software Safety Audit Reports are produced during the course of the SRS development, a summary report should also be produced.

Section Three. Roles and Responsibilities

**12** General

**12.1** Staff involved in the development, verification, validation and audit of SRS, including management, should be competent in the use for development of SRS of the methods and tools that are to be used.

Where a particular responsibility is allocated to a team, the Design Authority should demonstrate that the team, when taken as a whole, possess the required skills and experience by virtue of its composition from individuals who have different skills and experience. In this case, the roles and responsibilities of the team members should be defined clearly by the Design Authority in order to ensure that the right combination of skills and experience is applied to each activity in which the team is involved.

The extent to which the following factors are addressed should be taken into account when assessing and selecting personnel against the roles that they are to take in the project:

a) Familiarity with the standards and procedures to be used.

b) Knowledge of the legal framework and responsibilities for safety.

c) Knowledge and experience in safety engineering.

d) Experience of the application area.

e) Experience in software development.

f) Experience in the development of SRS.

g) Experience with the methods, tools and technologies to be adopted.

h) Education, training and qualifications.

The assessment should include consideration of the length, depth, breadth and relevance of the knowledge, experience and training. Training records may be used as a source of some of this information.

The assessment should take into account the characteristics of the SRS development, including the integrity required and the extent of novelty to be employed in application, design concept, development technology and procedures: The higher the level of integrity or the greater the novelty, the more rigorous should be the assessment.

Whilst continuity of staff is desirable it is recognised that this may not always be feasible. The factors above need to be investigated for any subsequent appointments. Project-orientated training should also be provided.

**12.2** No guidance.

**13** Design Authority

**13.1** No guidance.

**13.2** The Design Authority may subcontract the development of the SRS to another organization. This subcontractor is referred to in this Standard as the Software Design Authority. In cases where the hardware and software of the system are developed by a single contractor, the Design Authority effectively appoints themselves as the Software Design Authority.

**13.3** No guidance.

**14** Software Design Authority

**14.1** No guidance.

**14.2** No guidance.

**14.3** No guidance.

**14.4** No guidance.

**14.5** No guidance.

**14.6** The Software Design Authority certifies the software component of the system to the Design Authority in order that the Design Authority can have the necessary confidence and assurance to certify the safety of the system to the MOD.

**15** Software Project Manager

**15.1** In addition to the specific requirements of this Standard relating to the development of SRS, the Software Project Manager should employ general principles of good software project management.

**15.2** The Software Project Manager is the Software Design Authority's project manager for the SRS development project.

**15.3** No guidance.

**15.4** No guidance.

**16** Design Team

**16.1** No guidance.

**16.2** In addition to the specification, design and coding of the SRS, the Design Team will often be responsible for the generation and discharge of proof obligations, conducting preliminary validation and performing static analysis. The exact division of labour between the Design Team and the V&V Team will be determined by the Software Design Authority.

**16.3** No guidance.

**17** V&V Team

**17.1** No guidance.

**17.2** It is extremely important that V&V is carried out independently of design, both to preserve objectivity and to minimize pressure for premature acceptance. Such independence also introduces worthwhile diversity into the software production process. The V&V Team should be composed of personnel who are independent up to senior management level of others involved in the project. V&V will be carried out more efficiently if good lines of communication at working level exist between the Design Team and the V&V Team, but these should be set up in such a way that independence is not compromised.

**17.3** No guidance.

**17.4** The role of the V&V Team is to provide independent V&V of the correctness (and thus the safety) of the SRS. Usually the V&V Team will do this by performing dynamic testing of the SRS and by checking the static analysis of the SRS that is carried out by the Design Team. The V&V Team may also carry out additional analyses which complement those previously conducted. The precise division of such responsibility is not important provided that independence in the conduct or checking of the testing is demonstrated by the Software Design Authority.

**17.5**The checks carried out by the V&V Team may take the form of a full verification of the correctness of formal arguments. However, if peer review has been carried out and documented, the V&V Team may use this evidence to provide an adequate level of verification by checking the reviews that have been conducted previously.

**18** Independent Safety Auditor

**18.1** The Independent Safety Auditor may be part of the system independent safety audit team appointed to audit the safety of the overall system and, especially for projects involving large quantities of SRS, the role of the Independent Safety Auditor may be taken by a team. The Independent Safety Auditor should be able to assess the safety of the SRS free from any possible conflicts of interest. The necessary independence can best be achieved by using an independent company but an independent part of the same organization as the Design Authority or Software Design Authority may be acceptable if adequate technical and managerial independence can be shown at Director or board level. In order to avoid possible conflicts of interest, agreement on the choice of Independent Safety Auditor should be obtained from all the organizations whose work is to be audited.

**18.2** The Independent Safety Auditor should be appointed in the earliest practicable phase of the project; certainly no later than the project definition phase. There should ideally be continuity between the safety assessor for the system containing the SRS and the Independent Safety Auditor for the SRS development Typically, the Independent Safety Auditor or Independent Safety Auditor team leader will possess several years of experience of SRS and of its implementation in systems, experience in the relevant application area and a standing at least equivalent to that of a chartered engineer. Although it is anticipated and highly desirable that the Independent Safety Auditor will possess a formal qualification of this nature it is not a mandatory requirement of this Standard because an individual who might be acceptable for

**18.2** (Cont)

the role through their depth of knowledge and experience gained on previous projects may not hold such a qualification on paper. Other members of the team, if appropriate, should be selected to ensure that the team as a whole has adequate experience of the methods, tools and procedures that the Design Authority proposes to apply.

**18.3** The Software Safety Audit Plan should at least be checked to see if it needs to be updated at the start of each phase of the SRS development.

**18.4** The Independent Safety Auditor should be able to obtain information that they consider to be relevant to the Software Safety Case without encountering any hindrance or delaying tactics. Any disputes should be referred to the MOD PM.

**18.5** The Software Safety Audit Report may consist of a number of reports produced at different times or covering different aspects of the project.

**18.6** The Independent Safety Auditor (or Independent Safety Auditor team leader) endorses the SRS Certificate of Design to certify that the SRS has been developed in accordance with this Standard as qualified in the Software Quality Plan. Such endorsement does not release the Design Authority from any responsibility for safety of the SRS: The Independent Safety Auditor has no overall responsibility for the safety of the software.

**19** Software Project Safety Engineer

**19.1** The Software Project Safety Engineer may be the same person as the Project Safety Engineer for the system, referred to in Def Stan 00-56. In any event, the Software Project Safety Engineer should have good and direct communications with the Project Safety Engineer to ensure coherence of safety matters across the system/software boundary.

**19.2** The Software Project Safety Engineer should be able to refer to the Independent Safety Auditor if safety concerns cannot be dealt with within the project management regime.

**19.3** If the Software Project Safety Engineer signs the SRS Certificate of Design on behalf of the Software Design Authority then this additional endorsement is unnecessary.

## Section Four. Planning Process[1]

### 20 Quality Assurance

**20.1** In the UK this requirement will be met through ISO 9001 and TickIT certification. The ISO 9001 and TickIT certification should be regarded as a minimum software quality requirement for the development of SRS. In the absence of any higher levels of certification, the Software Design Authority should also consider the use of other means of demonstrating the suitability of the software quality system for SRS development, for example by applying a software engineering process capability assessment model.

**20.2** The Software Quality Plan should give details of the quality requirements for the SRS development and should detail the quality assurance activities. The Software Quality Plan may be part of, or may be an annex to, a combined software development and quality plan rather than being a stand-alone document. The content of the Software Quality Plan should be as detailed in **annex B.6**.

**20.3** The compliance to this Standard should be detailed in a compliance matrix which separately addresses each subclause of Part 1 of this Standard. Each declaration of compliance should be supplemented by an explanation of how compliance is to be achieved, if this is not obvious. **Annex D** defines the requirements for justification for non-compliance.

### 21 Documentation

**21.1** The objective of documentation is to provide a comprehensive, accessible record of the SRS and of its development process.

**21.2** The relationship between the documents listed in **annex B** is illustrated in figure 2. The Software Design Authority may decide, in conjunction with the Design Authority and the MOD PM, that the information would be better presented using a different contents list to that given in **annex B**. For example, a number of documents may be incorporated into a single document, provided, firstly, that the complete documentation set incorporates all the contents detailed in **annex B**, and, secondly, that there is easy traceability between the **annex B** requirements and their incorporation in the actual document set. However, it is important that the documentation structure chosen does not adversely affect the possible future need for the software to be maintained by another organization. The documentation should consist of a top-down hierarchy of documents, with each lower level being a more detailed representation of the previous level. Repetition of the same information at different levels of the documentation should be avoided.

**21.3** Precise details of the format and layout of documentation should be agreed between the Software Design Authority, the Design Authority and the MOD PM. Where there is conflict between this Standard and the chosen documentation standard, this Standard should take precedence. It is good practice in any software development for each plan to refer to all other relevant plans.

[1]NOTE: The contents of this Section should be read in conjunction with Def Stan 05-91 and 05-95.

**21.4** A complete and comprehensive documentation set is essential if, for any reason, the SRS development needs to be transferred to another organization, for example for in-service support and maintenance.

**21.5** In addition to identifying all software that has been developed during the development of the SRS, the documentation should also identify the precise versions of support software and hardware used.

**21.6** The appropriate time for documentation to be produced is during the activity to which the documentation relates. The Software Design Authority's procedures should require the documentation to be issued prior to commencement of the next dependent activity. Retrospective preparation is not acceptable.

**21.7** In particular, Def Stan 05-91 details the review of documentation. All documents in the document set are to be reviewed prior to issue.

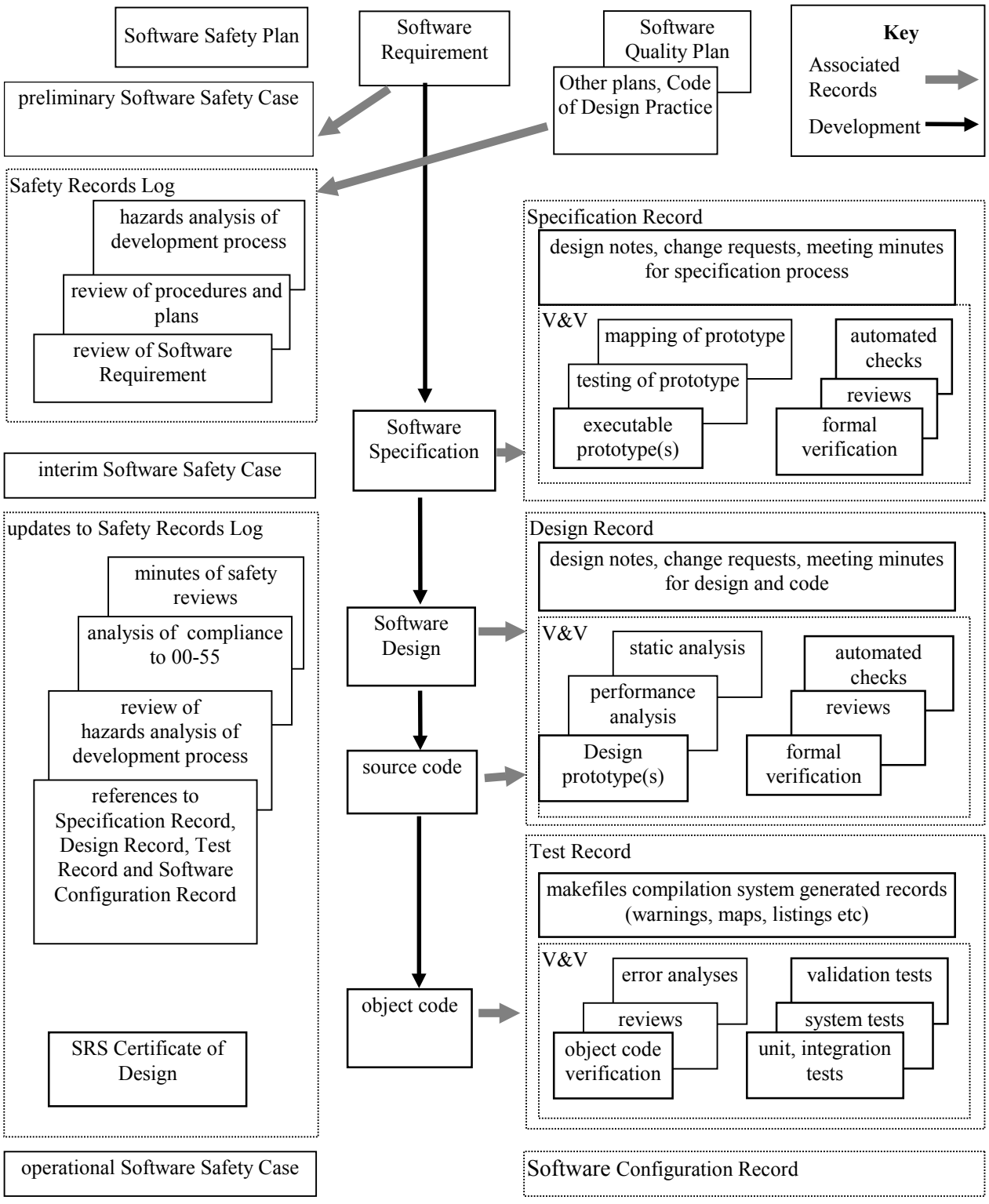



Figure 2: Software Development Records

## 22 Development Planning

**22.1** Development planning should address issues relating to the whole SRS development, including the process adopted, the inter-relationships between development processes, sequencing, feedback mechanisms and transition criteria.

**22.2** Experience shows that changes to requirements are likely at some point during the development of nontrivial software. In some cases there will be a conscious intention to change the requirements, for example where an evolutionary development process has been chosen. In other cases change may be necessary to take into account some unforeseen change in the environment in which the software is required to operate. Such changes may be small, and may not affect the safety requirements of the software, but it is important that the development planning recognises the possibility for change and identifies the process by which changes to the requirements will be accommodated and the safety of the software assured.

**22.3** Metrics are essential in order to be able to use past experience to accurately plan a new development. Metrics are also a useful indication of software quality and hence software safety. Relevant metrics include those associated with software development productivity and those associated with fault discovery. The identification of the metrics to be recorded should also indicate the values that are to be expected of each, and the interpretation that is to be made when the actual values differ from the target or expected values.

**22.4** The Software Development Plan should refer to the Software Safety Plan and the Code of Design Practice. Where the Code of Design Practice is general, the Software Development Plan should describe how the Code of Design Practice is instantiated for the particular development. The content of the Software Development Plan should be as detailed in **annex B.7**.

## 23 Project Risk

**23.1** For SRS development there is likely to be a connection between project and safety risk. This needs to be taken into account during any risk management activity. In particular, where actions are required to reduce a project risk (eg a slippage), it is necessary to ensure that there are no adverse effects on safety. The risk management process should involve a continual review of risks and the means to mitigate those risks such that control is maintained of all factors that may adversely affect the project.

**23.2** Where the Software Design Authority already possesses suitable procedures governing risk management the plan should define how the procedures are to be applied to the particular project. The Risk Management Plan should be examined to ensure that the items identified in **annex B.8** are included.

**23.3** Any implications from the risk analysis relating to software safety should also be communicated to the Independent Safety Auditor. Any changes to the SRS development as a result of project risk analysis should be assessed from a safety perspective within a software safety review (see clause **10**).

**24** Verification and Validation Planning

**24.1** No guidance.

**24.2** No guidance.

**24.3** The Software Verification and Validation Plan may be subdivided into separate Verification and Validation Plans for each phase of the SRS development. Separate plans may also be produced for individual V&V activities, such as static analysis or dynamic testing. In the case of decomposition in this manner an overview should be provided of the complete V&V process. The content of the Software Verification and Validation Plan should be as detailed in **annex B.9**.

**24.4** The acceptance criteria required to be detailed in the Software Verification and Validation Plan relate to acceptance internal to the SRS development. Acceptance in this context means that the particular item is complete, has been developed in accordance with the Code of Design Practice, has successfully undergone formal verification, static analysis and unit testing, and is ready for incorporation into the SRS. Procedures should be defined in the Code of Design Practice (see **annex B**) for deciding the extent of the rework necessary in cases where faults in the SRS have been found during development. Where such faults are within the scope of one of the V&V methods, but have been missed by that method, the procedures in the Software Verification and Validation Plan should require a re-appraisal of the safety analysis of the SRS development process and a reassessment of existing V&V results, possibly leading to a repetition of V&V activities throughout the SRS and the introduction of additional checks.

**25** Configuration Management

**25.1** The establishment and subsequent management of configuration control of the SRS is an essential element in maintaining the overall safety integrity of the equipment being developed.

**25.2** Def Stan 05-57 allows for two categories of Engineering Changes, namely Modifications and Amendments. Engineering Changes categorised as Amendments require no specific paperwork or approval and are intended for the correction of errors in software where the changes will not affect software function. This is not considered appropriate for SRS and hence the more rigorous form of Engineering Changes, namely Modification, is required. Alternative standards or procedures to Def Stan 05-57 may be used, as long as they conform to the requirements of Def Stan 05-57 and are agreed between the Software Design Authority, the Design Authority and the MOD PM.

**25.3** Configuration control is particularly important when SRS is being developed and the scope of the control will need to be greater than for other development projects. In particular it will be necessary to provide configuration control of the formal specification and all assurance activities (for example proof, static analysis, dynamic testing) associated with each configuration item. Whilst control of copies of the SRS following delivery is outside the scope of the Software Design Authority's configuration management system, it is important for the Software Design Authority to keep comprehensive records of the copies of the SRS that are delivered.

**25.4** No guidance.

**25.5** This will necessitate keeping records of the versions of the hardware, operating systems and software used to produce a particular version of the SRS and ensuring access to such hardware, software and operating systems if the need arises. It may be costly to recreate the SRS and its development environment from scratch, but on the other hand it may be costly to maintain the SRS and development environment at a level of readiness where reversion to previous versions is easy. The Software Design Authority should therefore balance the costs of maintaining easy reversion to a historical state against the likelihood of being required so to do.

**25.6** It may be impracticable to use the same configuration management tool for the control of all items. Ideally, however, all configuration items should be included within the automated configuration system.

**25.7** Configuration items are to be subject to configuration control as soon as they are given an identity and before they are referred to or used. It is not acceptable for configuration control to be applied retrospectively.

**25.8** The rigour of the configuration control of each software item should be commensurate with the criticality of each item. The identification of the safety integrity level of each software item should also help prevent software items of insufficient integrity being included in an application.

**25.9** The main safety risk to systems containing SRS is likely to be from accidental events. However, it is possible that there may be a risk from deliberate acts. Precautions, for example vetting of staff and restricted access to the software and data, should be taken commensurate with the level of risk.

**25.10** No guidance.

**25.11** The Software Configuration Management Plan may be part of the Configuration Management Plan for the system of which the software is part. The content of the Software Configuration Management Plan should be as detailed in **annex B.10**.

**25.12** The content of the Software Configuration Record should be as detailed in **annex B.11**.

**26** Selection of Methods

**26.1** Formal methods will be the primary means used for the specification and design processes of the SRS, but other methods (for example structured design methods) should be used as appropriate to complement the particular formal method chosen. In practical terms the use of formal methods is likely to involve certain gaps which cannot be bridged by formal arguments. It is possible to use formal methods at various points and in various ways during the SRS development process but, for non-trivial developments, it is not likely to be possible to produce a consistently rigorous and continuous formal development route from requirements to code. Gaps that cannot be bridged by formal arguments are most likely to arise where there is a major structural discontinuity in the SRS, for example where a subsystem is decomposed into a number of separate modules. In such areas structured design

**26.1** (Cont)

methods should be used to complement the formal methods and to provide the confidence in the transformation link.

**26.2** Required methods

**26.2.1** Formal methods

**26.2.1.1** Formal methods are mathematically based techniques for describing system and software properties. They provide a framework within which developers can specify, develop and verify software in a systematic manner.

**26.2.1.2** In order to be suitable for safety related applications, a formal method should be supported by industrialized tools and should contain a formal notation for expressing specifications and designs in a mathematically precise manner. (An `industrialized tool' is a software tool intended for use on a commercial software development and with a level of robustness that is no worse than that of a typical commercially available software tools. An industrialized tool would also possess commercial-quality user documentation, be usable by competent, appropriately trained, software engineers and the user would have access to effective support from the supplier. This could be contrasted with an `academic tool' which may only be suitable for use on small examples of code, may be unreliable and unsupported mad may only be usable by specialists.) This notation should have a formally defined syntax, should have semantics defined in terms of mathematical concepts and should be appropriate to the application.

**26.2.1.3** Other desirable features of a formal method include:

a) a means of relating formal designs expressed in its formal notation, preferably in terms of an associated proof theory for the verification of design steps;

b) guidance on good strategies for building a verifiable design;

c) documented case studies demonstrating its successful industrial use;

d) its suitability for design as well as specification, either on its own or in combination with another formal method;

e) the availability of courses and textbooks;

f) a recognized standard version which should preferably be an international or national standard or, failing that, a published draft standard.

**26.2.2** Structured design methods.

**26.2.2.1** Structured design methods comprise a disciplined approach that facilitates the partitioning of a software design into a number of manageable stages through decomposition of data and function. They are usually tool supported.

**26.2.2.2** A range of structured design methods exist covering a range of application areas, for example real-time, process control, data-processing and transaction processing. Many are graphical in format, providing useful visibility of the structure of a specification or design, and hence facilitating visual checking. Graphical representations include data-flow diagrams, state transition diagrams and entity-relationship diagrams.

**26.2.2.3** Many structured design methods possess their own supporting notation that in some cases may make use of mathematical notations thereby providing scope for automatic processing.

**26.2.3** In the context of this Standard, structured design methods are considered to include object orientated methods.

**26.2.4** Static analysis.

**26.2.4.1** Static analysis is the process of evaluating software without executing the software. For the purposes of this Standard, static analysis is considered to comprise a range of analyses which includes all of the following:

a) Subset analysis: identification of whether the source code complies with a defined language subset. Where the subset is not statically determinable this analysis will necessitate modelling the dynamic aspects of the software (eg through semantic analysis - see below) in order to check that the dynamic constraints of the subset are also met.

b) Metrics analysis: evaluation of defined code metrics, for example relating to the complexity of the code, often against a defined limit.

c) Control flow analysis: analysis of the structure of the code to reveal any unstructured constructs, in particular multiple entries into loops, black holes (sections of code from which there is no exit) or unreachable code.

d) Data use analysis: analysis of the sequence in which variables are read from and written to, in order to detect any anomalous usage.

e) Information flow analysis: identification of the dependencies between component inputs and outputs, in order to check that these are as defined and that there are no unexpected dependencies.

f) Semantic analysis/path function analysis (or symbolic execution): conversion of the sequential logic of the code into a parallel form to define the (mathematical) relationship between inputs and outputs. This can then be manually checked against the expected relationship, as defined in the detailed design, for each logical path through the component, in order to verify that the semantic logic of the code is correct. Semantic analysis can also be used to carry out checks against language limits, for example array index bounds or overflow.

g) Safety properties analysis: analysis of worst case conditions for any non-functional safety properties, including timing, accuracy and capacity.

**26.2.4.2** Static analysis is usually performed, using software tools, on source code to verify each software component against the lowest levels of the design. It may require annotation of the source code, typically in the form of tool-interpreted source language comments. Static analysis may also be performed:

a) on the design, to assist in the verification of one level of design against the preceding one, where the design is represented in an appropriate format;

b) to verify that object code is a correct refinement of source code, by statically analysing decompiled object code and demonstrating that this possesses identical semantics to the source code.

**26.2.4.3** Some static analysis tools are also able to verify or prove code against low level mathematical specifications or verification conditions. From the point of view of this Standard, these activities are considered to come under the category of formal arguments, although they are likely to represent only a part of the formal arguments necessary to show the conformance of object code to the Software Requirement.

**26.2.5** Dynamic testing

**26.2.5.1** Dynamic testing is the evaluation of software through the execution of the software. In the context of this Standard it comprises a `formal' part of the verification process, to be conducted only when there is a high degree of confidence that the software will successfully pass the tests. It is to be differentiated from debugging which is the process of executing the software to locate, analyse and correct suspected faults.

**26.2.5.2** Dynamic testing should be conducted at unit, integration and system levels to verify the code against the detailed design, architectural design and specification. It is typically supported by test tools to assist in the automation of the testing and the measurement of attainment of test targets, for example statement coverage.

## 27 Code of Design Practice

**27.1** The Code of Design Practice may be derived from an existing company manual containing the Software Design Authority's interpretation of the software development practices required by the Standard. Where sections of the Code of Design Practice are unchanged from a previous document they should be reviewed to confirm suitability for the current application and to ensure conformance with the requirements of this Standard. A newly developed Code of Design Practice will be reviewed in accordance with the requirement in clause 22 of Part 1 of this Standard for all documents to be reviewed. The Code of Design Practice should be updated as appropriate whenever refinements to the development methods are made. The content of the Code of Design Practice should conform to that defined in **annex B.12.**

**27.2** No guidance.

## 28 Selection of Language

### 28.1 High-level language requirements

**28.1.1** Definitions of the language characteristic terms `strongly typed' and `block structured' are provided in annex A of Part 1 of this Standard.

**28.1.2** A formally-defined syntax is one that is represented in an appropriate formal or mathematical language. A suitable formal language for representing programming language syntax is Backus-Naur Form (BNF).

**28.1.3** Currently all `real' languages have features which are undefined, poorly defined or implementation-defined. Furthermore, most languages have constructs that are difficult or impossible to analyse. Until a `safe' language has been developed it is inevitable that a language subset will be used for coding the SRS, in order to ensure that the program execution will be both predictable and verifiable.

**28.1.4** In order for program execution to be predictable, the semantics of the language need to be well defined, either formally or informally. Both static and dynamic checks are required to ensure that the requirements of the language hold. Static checks and enforcement of the static language properties should be performed by the compiler used. Checks of the dynamic properties should be performed by analysis or other means. The language checks should include checks for conformance to any defined language subset and will typically be performed during static analysis (see clause **26**).

**28.1.5** In order for program execution to be predictable, there should be a link (facilitated by the formally defined syntax) between the formal design and the execution of statements in the implementation language. This link should be provided by semantics for the language which form a basis for carrying out formal proofs about the code (the proof theory). The compiler that is used should be shown to be consistent with these semantics.

**28.1.6** The link between the formal design and the source code should be achieved by selecting an implementation language such that:

a) it has generally accepted formal semantics and a proof theory in terms of the formal method in use;

or

b) it has implicit formal semantics and a proof theory embodied in a proof obligation generator.

**28.1.7** In the absence of either of these options, the implementation language should be linked by hand to the formal method in use and a justification should be provided for its choice.

**28.2** Justification should also be provided in the Software Safety Case that the language checks performed by the compiler and by other means are adequate to ensure predictable program execution.

**28.3** Requirements covering coding practices are defined in clause **36**.

**28.4** No guidance.

**28.5** Use of assembler.

Experience shows that programming in assembler language is more error-prone than programming in a high-level language. There are circumstances, however, in which the practicalities of real time operation are such that the use of assembler is necessary in order to meet performance requirements. Furthermore, the use of a high-level language introduces some additional risks compared with the use of assembler. Specifically, high-level language compilation systems are more prone to error and less easy to verify than assemblers. There is therefore a need to trade off the risks of assembler programming against the risks of the use of a high-level language compiler, linker and runtime system. In general it is considered that only for very small amounts of assembler do the safety benefits of not having a compilation system outweigh the increased risk of programming error. Programming rules for assembler language programming should be more strict than those for high-level languages: Specific requirements are detailed in clause **36**.

**28.6** Compilation systems

**28.6.1** Where the compilation system is classified as being of the highest safety assurance level, there is a requirement, under selection of tools (see **29.5**), that the compilation system should be developed to the requirements of this Standard. However, the development of such a compiler is unlikely to be commercially viable and, in any case, the small user base of such a product would be a negative factor in justifying its safety.

**28.6.2** In practice the Software Design Authority should choose a combination of SRS design, compiler assurance and object code verification that is shown from the safety analysis of the SRS development process to maximize confidence in the correctness of the compilation process (see 36.6 of Part 1 of this Standard).

**28.6.3** Methods of providing assurance of the correctness of a compiler include:

a) Compiler validation: An internationally agreed black box method of testing compilers, designed to demonstrate that a compiler conforms to the appropriate international language standard. A validation certificate indicates that a compiler has successfully passed all the tests in the appropriate validation suite.

b) Evaluation and testing: The conduct of evaluation and testing beyond that provided by validation. Such evaluation and testing may examine the compiler in more depth than validation and is also able to assess the compiler in the particular configuration in which it is to be used on the SRS, which may not be the same configuration in which the compiler was validated.

c) Previous use: Evidence provided by successful previous use. This can be used as an argument for increasing the confidence in the correctness of the compiler. The requirements for configuration management, problem reporting and usage environments should be the same as for previously developed software (detailed in 30.5 of Part 1 of this Standard).

**28.6.4** The use of many programming languages will involve the use of an existing run-time system. This should be considered as use of previously developed software which is discussed in clause 30 of Part 1 of this Standard.

## 29 Selection of Tools

**29.1** Tools are essential in an industrial scale project as they allow processes to be performed which otherwise would be impracticable or inefficient and they allow the developer to have a higher degree of confidence in the correctness of a process than might otherwise be the case. In addition to any tools that are needed for the specific methods to be used, tools to support common functions such as configuration management, checking of specification and design documents, static analysis, dynamic testing and subsequent manipulation of object code are required for the efficient application of the Standard. Safety assurance, defined in annex A of Part 1 of this Standard, is a subjective measure of confidence in safety; as opposed to safety integrity which is an objective and quantified level.

**29.2** The safety assurance requirements for the tools used to support the various functions of the SRS development process are dependent on the contribution of the tools to the achievement of a safe product. In the same way that SRS should be considered in a systems context, the use of the tools should be considered in the context of the overall development process. The safety analysis of the SRS development process should therefore consider each tool in the context of its use in the SRS development.

**29.3** No guidance.

**29.4** The safeguards may consist of the use of a combination of tools or manual checks. Further guidance is provided in **29.5**.

### 29.5 Tool selection criteria

**29.5.1** Tools required to be of the highest level of safety assurance should be developed to the requirements of this Standard. It is anticipated that few tools, if any, are likely to be developed to this requirement, and hence diverse checks will almost always be required to reduce the safety assurance requirements for the tool to a manageable level.

**29.5.2** Where the safety analysis and tool evaluation indicates that a single tool does not have the required level of safety assurance, combinations of tools or additional safeguards, such as diverse checks and reviews, should be used. The independence and diversity of the methods or tools and their effectiveness in achieving the required level of safety assurance should be addressed in the safety analysis of the SRS development process and justified in the Software Safety Case.

**29.5.3** In situations where combinations of methods or tools are being used to achieve a higher combined safety assurance, human intervention may be involved in the combining function. Because human conduct of repetitive or monitoring tasks is error-prone, such a function should be as simple as possible and should be cross-checked. Alternatively the Software Design Authority may be able to achieve a higher level of safety assurance by the development and use of a simple tool.

**29.5.4** For lower levels of safety assurance the Software Design Authority will need to justify the adequacy of the development of each tool and evidence of correctness. Such justification, and arguments that the safety assurance of the tool is appropriate for the use of the tool in the SRS development, should be contained in the Software Safety Case.

**29.5.5** Where appropriate, validation and evaluation of tools should be carried out by an approved third party using a recognized test suite, acceptance criteria and test procedures, taking into account the operational experience with the tool. Where these services are not available, the Software Design Authority should undertake or commission an evaluation and validation and provide arguments that the tools meet a required level of safety assurance. The installed base, defect history and rate of issue of new versions of the tool are also useful factors, as is ISO 9001 and ISO 9000-3 conformance of the tool supplier.

**29.5.6** The Software Design Authority has the overall responsibility for selecting tools and should address the matching of tools to the experience of members of the Design Team as this has an important effect on productivity and quality, and hence on safety assurance. It should be appreciated that compromises are inevitable in assembling a coherent tool set from a limited set of candidates. As part of the determination of adequate safety assurance, consideration should also be given to the interaction between the Design Team members and the tool, and the level of safety assurance that can be given to this interface.

**29.5.7** The needs of support and maintenance activities during the in-service phase should be taken into account when selecting tools. In order to minimize support requirements, consideration should be given to the selection of a limited number of broad spectrum tools in preference to the selection of a larger number of specialized tools. Tools which are unlikely to have continued support through the life of the equipment should not be employed in the SRS development.

**29.5.8** Should it prove impracticable to define a SRS development process that can be supported by tools of the required levels of safety assurance, even after taking the above into consideration, a course of action should be proposed by the Software Design Authority and agreed by the Independent Safety Auditor and the MOD PM. In general, tools should be used as follows:

a) Whenever practicable a tool of the required level of safety assurance should be used.

b) Where it is not practicable to use a tool of the required level of safety assurance, a combination of tools should be used to provide the necessary added safety assurance.

c) Where it is not practicable to use a combination of tools, the use of a tool should be combined with an appropriate manual activity.

d) Only if there are no suitable tools available should a process be performed manually.

**30** Use of Previously Developed Software.

**30.1** When proposing to make use of previously developed software the following points should be considered:

a) The appropriate reuse of well proven software can be of substantial benefit to the integrity of SRS.

b) The policy for the use of previously developed software should be detailed in the Software Safety Plan which should also contain criteria for acceptability (eg appropriate in-service data for the previously developed software).

**30.2** Unreachable code. Unreachable code should be documented in the Software Design and justification for its retention should be provided in the Software Safety Case.

**30.3** Where software has been developed to the requirements of this Standard it is necessary to ensure that any differences between the original and new applications will not reduce the safety integrity of the code. Where differences exist, V&V activities should be repeated to a degree necessary to provide assurance in safety. For example, if a different compiler or different set of compiler options are used, resulting in different object code, it will be necessary to repeat all or part of the test and analysis activities that involve the object code.

**30.4** Previously developed software not developed to the requirements of this Standard

**30.4.1** Ideally all software that was not originally developed to the requirements of this Standard should be reverse engineered to a level equivalent to that of software that has been developed in accordance with this Standard. Reverse engineering in this context means the conduct of retrospective activities that include formal specification, formal design, formal arguments of verification, static analysis and dynamic testing. Access to the source code, design documentation and the software supplier is likely to be essential for this to be practicable. The primary aims of the reverse engineering are to provide an unambiguous and complete definition of what the software does and to verify, to an appropriate level of confidence, that the software meets this definition. A secondary aim is to produce documentation that will assist the incorporation of the previously developed software into the SRS and which will facilitate future maintenance. However, there may be cases where retrospective application of this Standard in its entirety to previously developed software would not be practicable. In such cases a reduction of the amount of reverse engineering activities may be possible if such a reduction can be justified from the safety analysis, taking into account the in-service evidence (see **30.5**). An example may be a run-time system or library where the software has been used without failure in diverse circumstances.

**30.4.2** In deciding the extent of reverse engineering, an evaluation should be made of the development process of the previously developed software, in particular its general adequacy and conformance to the requirements of this Standard. The reverse engineering activities should concentrate on the areas most notably inadequate in satisfying the objectives of this Standard. Specific issues to be addressed should include:

a) Conformance of the development process of the previously developed software to ISO 9001 and ISO 9000-3.

**30.4.2** (Cont)

b) Appropriate configuration management processes that provide traceability from the software product and lifecycle data of the previous application to the new application.

c) The methods and tools used during the development of the previously developed software.

d) The extent of the V&V activities conducted on the previously developed software.

e) The adequacy of the documentation for the previously developed software.

**30.4.3** The problem report history of the previously developed software should be analysed to identify the origin of the problems and thereby gain an indication of the adequacy of all or parts of the development process. The extent to which previously developed software that has not been developed to the requirements of this Standard may be incorporated in the new application should be agreed between the Software Design Authority, the Design Authority, the Independent Safety Auditor and the MOD PM.

**30.5** In-service history

**30.5.1** In order for in-service history of the previously developed software to provide any degree of objective evidence of integrity, there will need to have been effective configuration management of the previously developed software and a demonstrably effective problem reporting system. In particular, both the software and the associated service history evidence should have been under configuration management throughout the software's service life.

**30.5.2** Configuration changes during the software's service life should be identified and assessed in order to determine the stability and maturity of the software and to determine the applicability of the entire service history data to the particular version to be incorporated in the SRS.

**30.5.3** The problem reporting system for the previously developed software should be such that there is confidence that the problems reported incorporate all software faults encountered by users, and that appropriate data is recorded from each problem report to enable judgements to be made about the severity and implications of the problems.

**30.5.4** The operating environments of the previously developed software should be assessed to determine their relevance to the proposed use in the new application.

**30.5.5** Quantified error rates and failure probabilities should be derived for the previously developed software, taking into account:

a) length of service period;

b) he operational in-service hours within the service period, allowing for different operational modes and the numbers of copies in service;

c) definition of what is counted as a fault/error/failure.

**30.5.6** The suitability of the quantified methods, assumptions, rationale and factors relating to the applicability of the data should be justified in the Software Safety Case. The justification should include a comparison of actual error rates with the acceptability criteria defined in the Software Safety Plan and justification for the appropriateness of the data to the new application.

**30.6** No guidance.

**30.7** No guidance.

**30.8** No guidance.

**30.9** In cases where the previously developed software is justified on the basis of in-service history or extensive V&V and is treated as a `black box', it may be acceptable for design information not to be provided as long as a comprehensive requirements specification for the software is provided.

**31** Use of Diverse Software

**31.1** Software diversity is one of a number of techniques that may be used for design fault tolerance (see also **34.4**). The decision to use diverse software should be made as part of a coherent approach to fault tolerance, starting with system diversity covered in Def Stan 00-56, and following an objective analysis of the potential safety benefits and weaknesses of software diversity, including consideration of the following:

a) The aim of software diversity is to produce two or more versions of a software program in a sufficiently diverse a manner as to minimize the probability of the same error existing in more than one version. Consequently, when the diverse versions are operated in parallel, any design or implementation errors in any of the diverse versions should not result in common mode failure. Other names for software diversity are multiple-version dissimilar software, multi-version software, dissimilar software or N-version programming. Software diversity may be introduced at a number of stages of development, specifically:

i) design: typically through the development of each diverse version of the software by separate development teams whose interactions are managed; the development being conducted on two or more software development environments;

ii) coding: the implementation of the source code in two or more different implementation languages;

iii)compilation: the generation of object code using two or more different compilers;

iv) linking and loading: the linking and loading of the object code using two or more different linkage editors and two or more different loaders.

b) Diversity may be introduced at all of the above stages or in a more limited manner if there are specific development risks that are required to be addressed.

**31.1** (Cont)

c) Software lifecycle processes completed or activated before dissimilarity is introduced into a development remain potential error sources. This is particularly the case where each diverse implementation is derived from the same Software Requirement. Since experience indicates that the specification process is the largest single source of software faults, it is important that the requirements of this Standard are met in all aspects relating to the software specification.

**31.2** Some reduction in the rigour of the V&V processes on diverse components is possible if it can be demonstrated that the improved safety integrity from diversity outweighs the potential hazards of reducing the rigour of the SRS development process. The use of diverse software components with reduced V&V processes should be shown by the safety analysis of the SRS development process to have an equivalent or higher safety integrity level than a single development with full V&V activities. For example, by using diverse implementation languages and compilers the safety analysis may show that the risk of compiler errors causing hazards is reduced by such a degree that the amount of object code checking may safely be reduced.

**31.3** It should be noted that the use of diverse software can introduce problems, for example in the areas of synchronization and comparison accuracies between diverse components operating in parallel, which may increase the safety risk. It is also often difficult to judge the independence of the diverse components and hence to guarantee their freedom from common mode failures, particularly where common solutions to difficult areas of the problem domain may have been adopted. This makes the benefits of software diversity hard to quantify. Such issues should be addressed during the safety analysis and taken into account in the justification for both the use of software diversity and any reduction in the rigour of V&V activities.

## Section Five. SRS Development Process

This Section provides a set of techniques to be used in the development of SRS and describes what shall be done to claim conformance to each technique. The application of these techniques across various safety integrity levels is given in the Tailoring Guide shown in **annex D**.

### **32** Development Principles

#### **32.1** The lifecycle for the SRS development

**32.1.1** The lifecycle for the SRS development which is reflected in the structure of this Standard is illustrated in figure 3 and consists of the following development processes.

a) production of a Software Specification (see clause **34**);

b) development of a series of increasingly detailed Software Designs (see clause **35**);

c) coding the SRS (see clause **36**);

d) testing and integrating the SRS (see clause **37**).

**32.1.2** Decompositions of the lifecycle for the SRS development in terms of alternative development processes may be acceptable, provided that they can be related to those specified in this Standard. For example, coding and compilation might be considered as two development processes.

**32.1.3** The SRS development process should be defined in terms of its constituent development processes, each of which should have defined objectives, inputs and outputs. Each development process should be broken down in a similar way to arrive at a set of basic tasks which are sufficiently simple to be well understood in terms of the information content of their inputs and outputs and the definition of how the task is to be performed.

**32.1.4** Diagrammatic representations should be used in documenting the SRS development process as these are often helpful in communicating an overview of the development and verification processes and their relationships. Such diagrams can also be helpful when performing a safety analysis of the SRS development process in accordance with 7.4 of Part 1 of this Standard.

**32.1.5** It is assumed in this Standard that the development processes are phased sequentially, such that each is completed before the following one is started. However, it is acceptable for other phasing to be adopted provided that such phasing is well defined. One satisfactory way of doing this is to specify the entry and exit criteria for each development process. A sequential set of development processes is then one in which no development process can commence until the preceding development process has completed and partial overlap can be obtained by permitting a development process to start when a specified subset of the products of a previous development process are available. In order to provide a satisfactory level of control and to avoid extensive rework, the exit criteria for each development process should require the completion of all preceding development processes. In any case the verification of

**32.1.5** (Cont)

the outputs of a development process cannot meaningfully be carried out before the verification of its inputs.

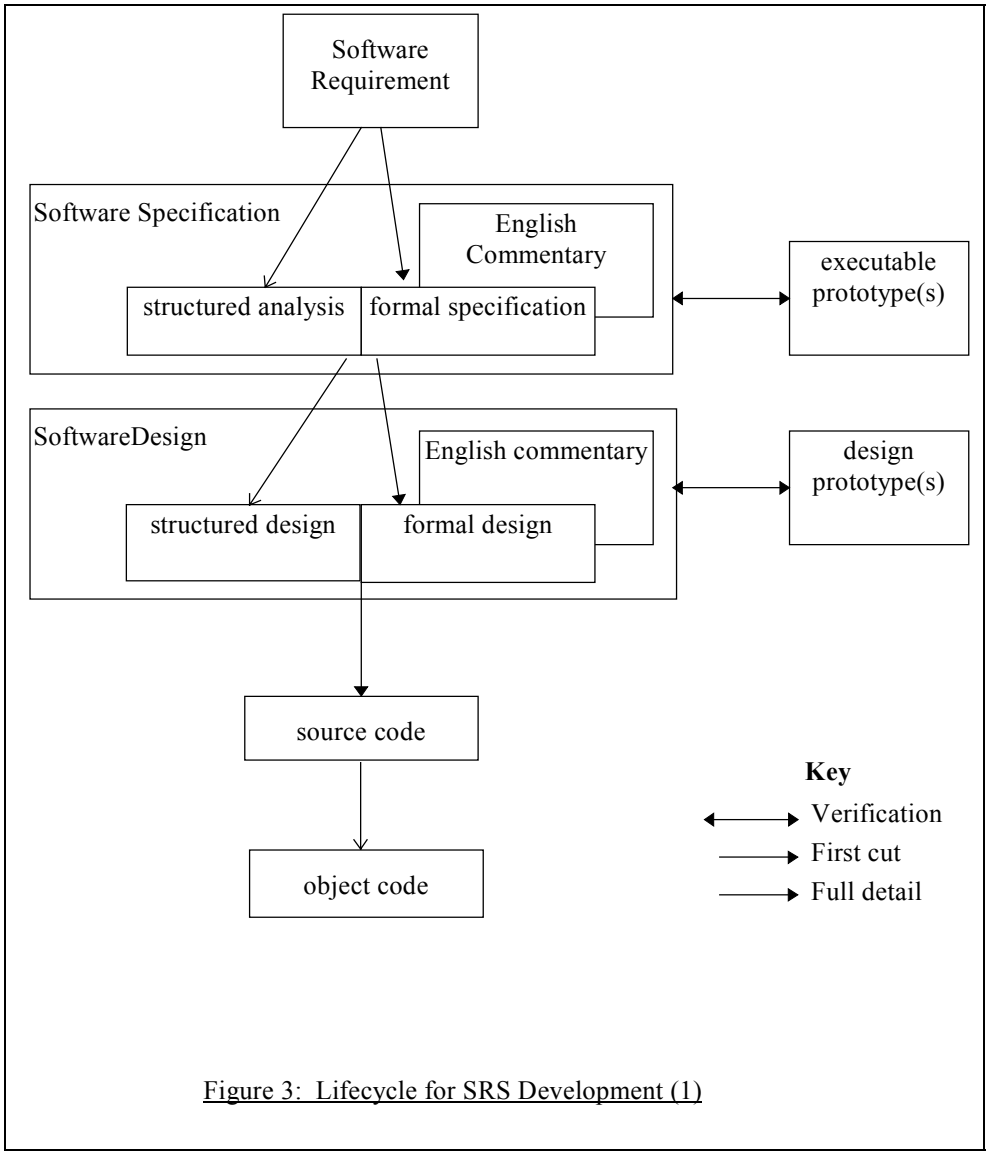



Figure 3: Lifecycle for SRS Development (1)

**32.1.6** The philosophy adopted by this Standard is that correctness should be achieved during development and emphasis should therefore be placed on avoiding the introduction of errors rather than on their subsequent elimination. The inputs and outputs of each development process should be checked both for internal consistency and for consistency with one another. Tests should also be devised at each stage so that it can be verified that the software performs as required. It follows that the context of each development process is as illustrated in **figure 4**.

**32.1.7** A top-level view of the lifecycle for the SRS development envisaged in this Standard is shown in **figure 5**. This figure should be viewed as illustrative of the overall approach and does not include detail, such as feed-back, which will occur (and is to be encouraged) in any real project. It should be stressed that the particular lifecycle that is adopted is not important provided that the following objectives are supported:

a) It should be possible to relate the development processes and tasks to the requirements of this Standard.

b) A progressive refinement from requirements to product should be visible.

c) There should be traceability of requirements through the documented outputs of the software development lifecycle.

d) There should be verification of the outputs of each development process.

**32.1.8** The role of each development process and task in meeting the requirements of this Standard should be clearly understood so that the contribution that each task makes to the overall development is documented and conformance with this Standard can be demonstrated. It is particularly important that this information is available if changes to the development process have to be made at a later stage.

**32.2** Production of the Software Specification, Software Design and code

**32.2.1** This Standard requires that the documentation for the software provides a series of views of its functionality and performance, from an initial high-level, abstract form (what is required) to a detailed implementable form (how this is achieved). When formal methods are used throughout, this takes the form of a formal specification which is related to a formal design and thence to the code via a series of refinement proofs. The documentation should assist the reader to see the high-level overview as well as the detail from both the Software Specification and Software Design (which may consist of more than one stage). The final stage of the design is the production of the source code from which is generated the object code that is implementable on the target hardware.

**32.2.2** The view taken in this Standard is that the clearest, most concise and least ambiguous specification and design will be achieved through the use of formal methods. To make the use of formal methods practical, the following points should be noted:

a) To assist in the development and assessment of the Software Safety Case, the Software Design Authority should ensure that all aspects relevant to safety can be readily identified.

b) Structured design methods can help in producing well structured and hence maintainable, understandable software. The view taken in this Standard is that formal methods and structured design methods are complementary and may be used in conjunction to produce a better product. Conventional (informal) methods should be used to structure and analyse the functionality. Informal methods are generally designed to be easy to use, are accessible and have good tool support. Formal methods are used to remove ambiguity and to enable mathematical reasoning to take place. If timing is critical to safety, then formal expression of the timing requirements will be necessary. For simple requirements, this can sometimes be achieved by representing time as a variable within the formal representation.

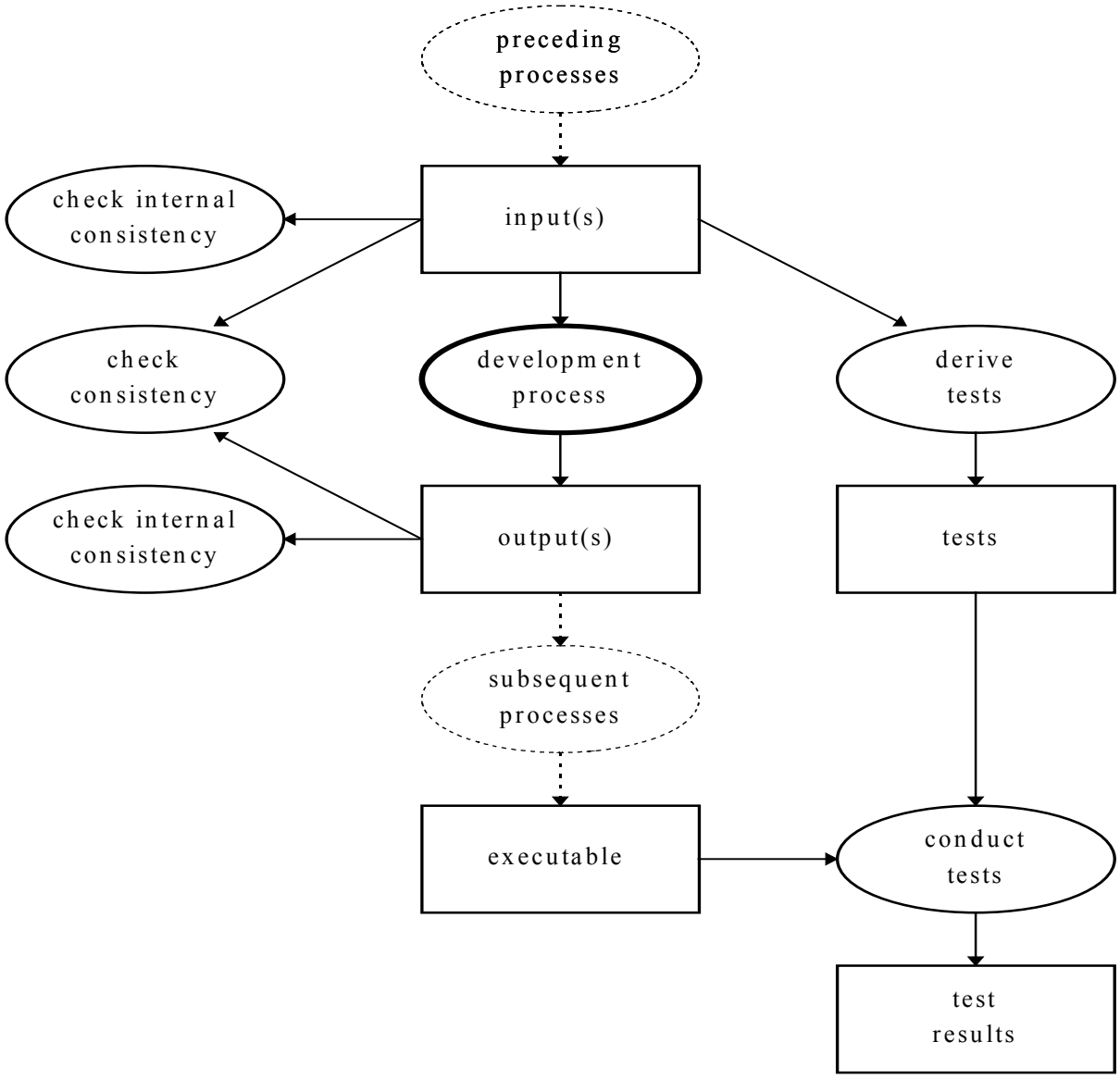



Figure 4: Context of a Development Process

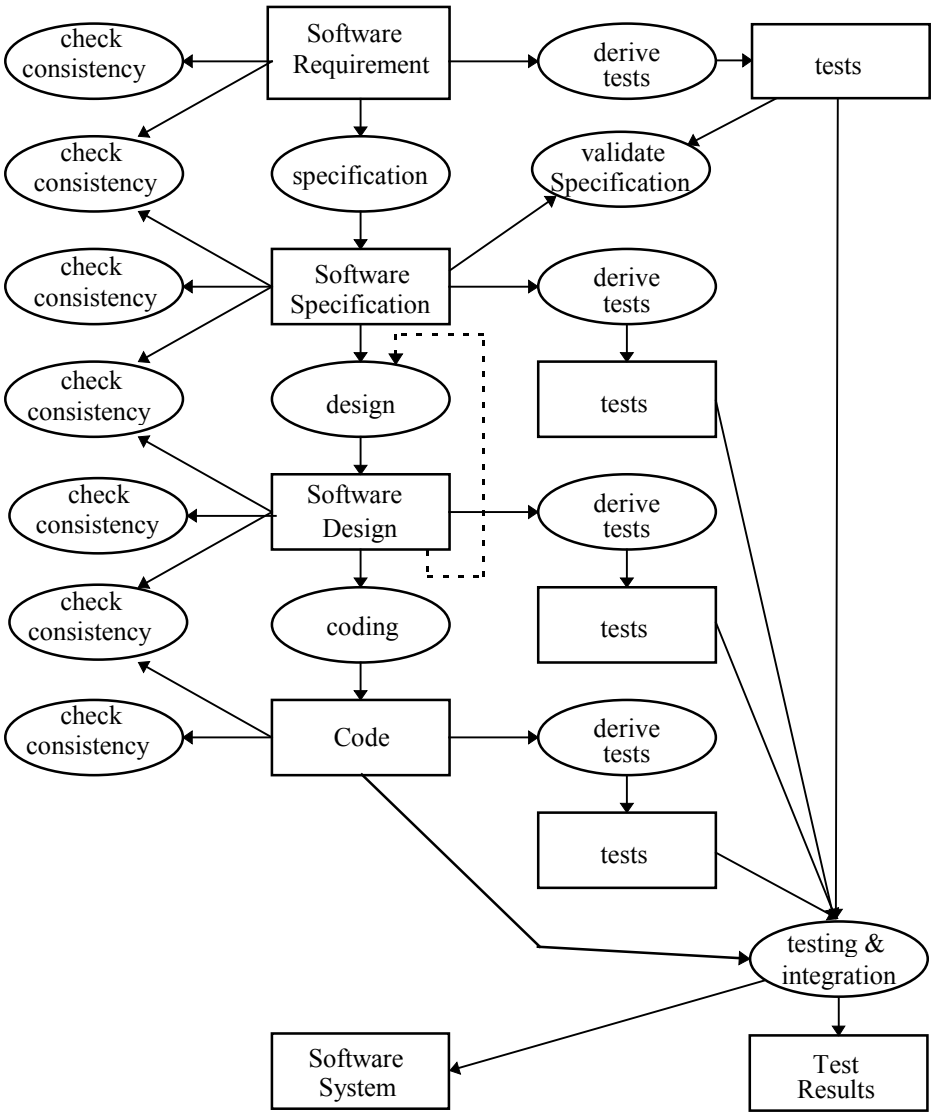



Figure 5: Lifecycle for the SRS Development (2)

**32.2.2** (Cont)

c) Formal methods should normally be used to represent the entire functional requirements of the SRS. However, there may be cases, agreed in advance between the Software Design Authority, the Independent Safety Auditor and the MOD PM, where the use of formal methods for the entire SRS would not contribute materially to the safety of the system. In these cases, the minimum requirement is that the safety functions and safety properties are represented formally.

d) If the formal specification does not represent all requirements, then it will rely to some extent on informal techniques. The Software Design Authority should ensure that the formal representation is sufficient to represent the safety features and that the informal parts which may be required to link it together do not introduce ambiguity. Reducing the scope of formal

**32.2.2** (Cont)

methods in the specification and design may result in an increase in the requirements for testing. In addition, regardless of the scope of the formal specification and design, the entire SRS code should be subject to static analysis to ensure that an error in the non-formally specified parts cannot introduce errors affecting the formally specified parts. Where the software relies on input of a particular form, such as configuration data, look-up tables etc, the characteristics and properties of this data should be formally specified and designed. This might include specifying type, range, accuracy, permissible values, relationships between two or more data items, permissible operations etc. It may not be possible to generate proofs of the safety properties without such specification,.

e) Even when all functional parts are formally specified, there will almost certainly be requirements for which formal methods are not well suited (such as maintainability, user-interface, reliability etc). These should be addressed by conventional methods. The Software Design Authority should ensure that using more than one notation does not lead to ambiguity, to conflict between two or more notations or to parts of the specification not being fully covered by any method. The interface between the methods should be documented in the Code of Design Practice, but if there are any specific areas where confusion might arise, an analysis of the interface should be provided in the Specification Record or Design Record.

f) The usefulness of the formal notation is considerably enhanced by an English commentary, which should help to provide the link between the mathematical notation and the physical conditions and objects that they represent. English commentary should be in the terms used by the system designers and should avoid mathematical `jargon'.

g) Tools should be used to assist in the preparation of the formal representations.

**32.2.3** The Specification Record, Design Record and Test Record should contain or reference all the quality assurance and verification records which are accumulated during the development and verification of the Software Specification, Software Design and code. The Software Design Authority may choose to have different document structures, but the link between the records and the processes within the SRS development process should be clear.

**32.2.4** The Specification Record relates to the specification of requirements, including the review of the Software Requirement and preliminary validation of the Software Specification. The Design Record relates to their implementation in the Software Design and code, including the static analysis and formal proof. The Test Record contains testing documents (including test specifications and results). Object code verification should normally be recorded in the Design Record, but object code verification performed by testing may be recorded in the Test Record.

**32.2.5** Many of the clauses in this Standard, are intended to improve the maintainability of the SRS, as well as the probability of developing it correctly in the first place. The SRS development is likely to be required to conform to the requirements of an integrated logistics support standard such as Def Stan 00-60. The Design Team should consider the lifetime over which the SRS will be required to be supported and, as this is likely to be of long duration, should ensure that the whole of the documentation is understandable to people outside the original team. Recording of assumptions and reasoning behind design decisions will help to

**32.2.5** (Cont)

ensure that future modifications can be made correctly. Consideration of maintainability also affects the choice of methods, languages and tools.

**32.2.6** Prototypes may usefully be used at several points in the development, for example to clarify requirements or to check feasibility of a design solution, but it is not acceptable for prototype code to become part of the SRS by the retrospective application of quality procedures. Once the prototype has been used to clarify the requirements or check the feasibility of the design, the code should be developed according to the development principles. Designers and coders are free to use such prototypes as a model for the implementation, but the design should not be reverse engineered from prototype code. Note that, except where it could not affect safety, prototype code does not satisfy the requirements for reuse of previously developed software (see clause **30**).

**32.2.7** To protect against inadvertent confusion between similar parts of prototype and formally developed code, the configuration management system should make a clear distinction between these categories (for example by using a different prefix in the naming convention), and this distinction should be made in the individual code items (for example in the module names used and in module headers).

**32.3** Traceability

**32.3.1** Traceability is required to assist in demonstrating that all requirements are satisfied and that the SRS does not perform in any way that is not specified. Traceability is also important in showing that the V&V is adequate. Traceability is important when changes are made to requirements, design or code, because the traceability helps to locate where the changes should be made and what tests should be repeated.

**32.3.2** It should be possible to trace each requirement through the Software Design to the part of the code which implements it, and it should be possible to trace from any part of the code, back through the Software Design and Software Specification, to the requirement in the Software Requirement which is satisfied. To a certain extent, the particular software development methods used will determine what is actually identified in the traceability system. It is likely however, that the functions, data and constraints will be recognizable in some form in the Software Requirement, and that therefore these should be identifiable in the Software Design and code even if they are packaged in some other manner.

**32.3.3** Traceability is also required for all forms of verification. This enables checking of the extent of the verification, and also helps to ensure that the extent of reverification in the event of changes can easily be assessed. Individual tests, or sets of tests, should be given traceability to the requirements and designs which are being tested. The philosophy behind each test should also be stated, including whether the test has been designed using white box or black box techniques and also the particular test technique or criteria which has been used. For example, one test might be a black box test designed to cover a particular equivalence partition; another, a white box technique added to achieve a particular coverage metric. Formal arguments should also be given traceability to the requirements and proof obligations which are being discharged.

**32.3.4** Traceability is normally achieved through the use of traceability matrices, or through the use of a traceability tool.

**32.3.5** However complete the Software Requirement, there will be times when the choice of implementation leads to additional derived requirements. For example, the division of the software system into smaller subsystems would result in interface requirements; the choice of a particular microprocessor and memory unit will result in specific timing and capacity requirements; the decision to write defensive code may result in the source code having a need to report particular failures via failure codes. The Software Design Authority should ensure that the eventual users and maintainers of the SRS have visibility of relevant requirements. This would normally be achieved by the Software Design Authority reporting such requirements to the Design Authority, who will have the responsibility of creating the user and system maintenance documentation.

**32.4** Verification of the Software Specification, Software Design and code

**32.4.1** The formal specification and formal design are mathematical objects constructed according to certain formal rules. Because of this, it is possible to carry out checks on them, in addition to simple manual review, and because of their mechanical nature many of these checks can be automated. Automated checks have the advantage of being repeatable and consistent. Syntactic checks are those which detect incorrect use of the notation. Semantic checking detects the presence of errors, such as type errors in syntactically correct specifications and designs. These checks resemble, but are generally more extensive than the checks provided by a good compiler for a high-level language.

**32.4.2** Where structured design methods have been used in the preparation of the Software Specification and Software Design, tool-supported syntax and consistency checks should be performed on these parts of the documents. Other parts of the documents, such as the English commentary can only be reviewed.

**32.4.3** The proof obligations for a particular formal method are the properties that the designer is obliged to discharge in order to have assurance that the specification is self consistent, or that a design correctly implements a specification (refinement). The proof obligations for self consistency and sometimes for refinement may be obtained from the published definition of the formal method. The goal of detecting inconsistencies in the specification is usually achieved by some form of satisfiability proof, ie proving that for all valid inputs, the output satisfies the requirements. Refinement proofs are required to verify the first stage of the design against the specification and to verify each subsequent design stage against the previous one. Manual generation of proof obligations is an extremely arduous and error prone task and a more assured method is to use a proof obligation generator. This is a tool which embodies the proof theory for the method.

**32.4.4** Proof obligations are discharged using formal arguments. Formal arguments can be constructed in two ways: by formal proof or by rigorous argument.

**32.4.5** A formal proof is strictly a well formed sequence of logical formulae such that each formula can be deduced from formulae appearing earlier in the sequence or is one of the fundamental building blocks (axioms) of the proof theory. Formal proofs are often highly intricate and very long, and may be difficult to read or write without tools. Tools should be

**32.4.5** (Cont)

used to assist in the generation of formal proofs and checking of formal proofs. Where tools have been used in the production of formal proofs, the proof output should be in a form suitable for use as input to other tools (eg proof checkers).

**32.4.6** A rigorous argument indicates the general form of a formal proof, by describing the main steps. It can be converted to a formal proof by the addition of detail and this process should not require creative steps. Since rigorous arguments provide a lower level of assurance than formal proofs, but are easier to produce, the decision should be made at the start of the project about the circumstances in which rigorous argument will be acceptable. This should be agreed with the Design Authority, the Independent Safety Auditor and the MOD PM. Agreement to any deviations, and to the detail of compliance with this policy, should be obtained as the formal arguments are created. Rigorous arguments should be stored in machine readable form and should be made available in this form for update, review and verification, in case it is thought necessary to add detail to convert a rigorous argument to a formal proof.

**32.4.7** Traceability should be provided between the formal arguments and the Software Requirement to assist reviewers of the formal arguments to check that all requirements have been verified, and to ensure that the implications of changes to requirements can be assessed.

**32.5** Review

**32.5.1** Review of formal verification

**32.5.1.1** In some organizations the Design Team will have the responsibility of generating and discharging the proof obligations, as part of the specification and design process. In other organizations, the V&V Team will have the responsibility for the construction and discharge of the proof obligations. However the work is allocated, the V&V Team either produces and discharges the proof obligations or reviews them. This ensures that there is an independent scrutiny of the Software Specification and the Software Design. In addition there should be at least a peer review of the proof obligations and formal arguments, so if these are produced by the V&V Team, members of the V&V Team other than the author should review them. The Code of Design Practice should record whether reviews should be exhaustive or use spot-checks. Some projects will require a mixture, with key proofs being exhaustively checked and others being subject to spot-checks only.

**32.5.1.2** Reviewers of formal proofs should use a tool. The tool used to check a formal proof should be different from the one used during its generation.

**32.5.1.3** Theorem prover tools are generally large and complicated and it is therefore difficult to ensure that they are free of errors. In contrast, theorem checkers can be much simpler. By using a different, and simpler, tool to check the proof the chance of an error being introduced by one tool and overlooked by the checking tool can be made very small.

**32.5.1.4** Rigorous arguments can only be manually checked. If there is any doubt about the rigorous arguments, attempts should be made to turn them into formal proofs.

**32.5.2** Review of Software Specification, Software Design and code

**32.5.2.1** This Standard requires that every output of the development process is reviewed. The timing of the reviews should be chosen to facilitate progress and minimize rework. It will probably be more effective to review the Software Specification and the results of automatic type and syntax checks before undertaking other verification processes. Similarly the Software Design should be reviewed before refinement proofs are performed, and the code should be reviewed before static analysis is performed.

**32.5.2.2** It is envisaged that the bulk of the reviewing will be done by other members of the Design Team. The V&V Team should be satisfied with the amount and the rigour of the review and should be satisfied that there is sufficient independence to detect errors and to assess the maintainability of the documents. A suitable compromise may be to include a member of the V&V Team in reviews of the completed outputs.

**32.5.2.3** The Software Design Authority should follow their own procedures in how reviews are organized, but the procedures in the Code of Design Practice should state:

a) the preparation time for the review;

b) requirements for the appropriate division of review material to ensure that the amount covered at each review is manageable;

c) procedures to ensure full coverage for all aspects of the development;

d) the roles and qualifications required of members of the review team;

e) the procedures for recording and resolving any problems found.

**32.5.2.4** The review should not attempt to perform the verification which is required in other clauses, but it should consider the results of verification.

**32.5.2.5** The review should also consider the adequacy of the verification, particularly with respect to the need to ensure that the software meets the safety requirements.

**32.5.3** Review of anomaly correction. Whenever an anomaly is discovered by any verification process, the Software Design Authority should first determine whether it constitutes an error in the SRS or is simply an artefact of the design or verification method. If it is an error, the Software Design Authority should attempt to determine how the error occurred, why it was not found sooner and whether there are likely to be other errors of the same type. If it is not an error, then an explanation and justification of the anomaly should be recorded in the Specification Record, Design Record or Test Record. Unnecessary changes should not be made to the SRS, so an evaluation of the effects of changing the SRS or of leaving it as it is should be made. The evaluation should address whether an uncorrected anomaly could lead to errors in the future. If the anomaly is an error, but it is decided that a correction will not be implemented, the details of the error, and justification for not correcting it shall be included in the Specification Record, Design Record or Test Record. This should also be referenced in the Software Safety Case. Once an error has been fixed, it is usually not sufficient simply to repeat the verification, as it is likely that the correction of the error will

**32.5.3** (Cont)

have been made in a manner that ensures that it passes the verification that detected the error. The verification should therefore be reviewed, and additional verification performed. If the error has escaped detection by earlier verification processes, it may be necessary to repeat earlier verification steps more rigorously, and to review and improve the Code of Design Practice. All investigations into errors should be recorded in the Specification Record, Design Record or Test Record as appropriate.

## **33** Software Requirement

### **33.1** General

**33.1.1** The purpose of the Software Requirement is to define a set of requirements which, if met, will ensure that the SRS performs safely and according to operational need. The Software Requirement is typically a natural language document which includes engineering notations such as diagrams and mathematical formulae. The Software Requirement should be derived from the requirements of the overall system, including safety, functional and non-functional requirements. Derived requirements may also arise during decomposition of the system into subsystems.

**33.1.2** The Software Requirement may form the technical part of a procurement specification. The whole of the subsequent SRS development process depends on the requirements being correct and well understood and therefore it is worthwhile to invest some effort in the requirements verification to ensure that the Software Design Authority is satisfied that the requirements are suitable for subsequent development. The review should consider whether the Software Requirement fully addresses the following:

a) software safety integrity requirements;

b) safety functions;

c) configuration or architecture of the overall system as far as this affects the SRS;

d) capacity and response time performance;

e) all interfaces between the SRS and other equipment or operators;

f) all modes of operation of the system in which the SRS is required to operate, in particular safety-related, maintenance, system testing and failure modes;

g) measures to overcome failure modes of the system,

hardware or software, (ie fault detection and fault tolerant mechanisms) which are to be implemented in software;

h) any safety related or other relevant constraints between the hardware and the software;

i) requirements for software self-monitoring, and requirements for the software to monitor actuators and sensors;

**33.1.2** (Cont)

j) requirements for periodic on-line and off-line testing of the system;

k) areas of functionality which are likely to change;

l) background information to enable the Software Safety Case to summarize the equipment level design approach to safety and describe the role of the software in ensuring safety.

**33.1.3** The Software Design Authority should also consider whether the Software Requirement:

a) is clear, precise, unequivocal, testable and feasible;

b) uses terms which are unambiguous and will be understood by all users of the document;

c) distinguishes safety functions from non-safety functions;

d) distinguishes between requirements for the product and those for the development process;

e) is quantitative with tolerances;

f) avoids constraining the design of the SRS unnecessarily.

**33.1.4** The input of the Software Design Authority can provide valuable insights, but is necessarily limited by the scope of the requirements and visibility of the context of the software in the system. The Design Authority and the Software Design Authority should be in agreement over the content and interpretation of the Software Requirement.

**33.2** A contractual boundary may exist with the Software Design Authority being provided with the Software Requirement. Since safety is a system property, only the Design Authority has the necessary information to decide the safety integrity level required of the SRS. However, the Software Design Authority may have previous experience in similar software systems, and should be satisfied that the safety integrity level demanded is both achievable and reasonable for the system.

**33.3** No guidance.

**34** Specification Process

**34.1** The role of the Software Specification is to provide a clear, concise, unambiguous specification of the software. This Standard requires the production of a Software Specification, consisting of a formal representation of the safety functions and constraints of the SRS, with other functions and constraints represented either formally or informally (see **figure 3**). The Software Specification should be structured using conventional structured design methods and should be supported by an English commentary. In addition, the development of the Software Specification should be supported by a Specification Record, which contains all the records pertaining to the development and checking of the Software Specification.

**34.2** Verification of the Software Specification

**34.2.1** The general method of constructing proof obligations and discharging them using formal arguments should be used to verify the Software Specification. Since the Software Requirement is informal, the proof obligations will be those required for internal consistency of the Software Specification. These should include proof that constraints demanded by the Software Requirement are satisfied by the Software Specification.

**34.2.2** All safety functions and all safety properties should be formally specified. Particular care should be taken to ensure that, for example, all timing and capacity constraints have been identified, and that constraints and relationships between data items are recorded.

**34.2.3** All checks, reviews etc should be documented in the Specification Record, which should also contain any completed checklists or other `working' documentation which indicates that the checks were performed.

**34.3** Preliminary validation

**34.3.1** This Standard allows for two methods of validating the Software Specification.

**34.3.2** Formal parts should be validated formally, ie by performing proofs that certain properties of the Software Specification hold.

**34.3.3** Non-formal parts and non-functional parts (including dynamic properties) are difficult to validate in this way, and will be more amenable to validation by means of one or more executable prototypes or by means of animation tools. Executable prototypes will also be useful to assist users who are not trained in formal methods to understand a formal specification. If the SRS interacts directly with users, prototyping the user interface is very important.

**34.3.4** The review of the Code of Design Practice within the software safety review provides the opportunity to decide if the proposed approach to preliminary validation is adequate.

**34.3.5** The preliminary validation should provide the opportunity to check that the specified system really will satisfy the operational requirements of the system and especially of the users. The end users of the system (or an informed representative), as well as specifiers of other systems which interface to the SRS should therefore be involved. The Design Authority will generally need to be involved in preliminary validation.

**34.3.6** Most projects have changes to the requirements during or after development. The Software Design Authority should define how such changes will be handled, and how much of the preliminary validation will be repeated if the requirements (or the interpretation in the Software Specification) change. In some cases, the requirements will evolve during the project, and it may be necessary to separate core functionality which can be formally specified and verified from the start, from evolutionary functionality which is prototyped until the requirements have stabilised. In some projects, such as those developing control systems, it may be necessary for the executable prototype to consist of code for the target system which is used by the system designers to help to define the software requirements.

**34.4** Preliminary validation via formal arguments

**34.4.1** Formal arguments should be used to explore and assess the completeness and consistency of an implementation deriving from the Software Specification, and in particular to identify:

a) incomplete functions;

b) poorly defined functionality (for example specifications that do not cover all expected cases);

c) errors in the Software Specification that lead to inconsistent states, failure conditions or erroneous results.

**34.4.2** Formal arguments should also be used to:

a) demonstrate that undesirable action does not occur;

b) establish the behaviour at the boundaries of the Software Specification;

c) explore and assess extensions and improvements that can be obtained without extensive modification by building on what is already specified.

**34.5** Preliminary validation via executable prototype

**34.5.1** Executable prototypes can be developed by a variety of means, such as:

a) translation into a suitable executable language;

b) animation of the Software Specification by means of a tool designed for that purpose;

c) use of an executable subset of the specification method.

**34.5.2** The Software Design Authority should try to ensure that the changes made in converting from the Software Specification to the executable prototypes are kept to a minimum and are mathematically justifiable in that they preserve the properties of the Software Specification. The Software Design Authority is required to construct and discharge proof obligations which demonstrate the link between the Software Specification and the executable prototypes. In the cases where a tool has been used to perform the translation or animation, the tool vendor should be able to provide assistance in explaining and justifying the transformations used. To minimize the effort involved, the Software Design Authority should try to make the arguments as general as possible. Rigorous argument, rather than formal proof should normally be sufficient for discharge of the proof obligations.

**34.5.3** If an executable prototype covers elements of functionality that are not included in the formal specification then there should be an explanation of how the non-formal parts of the Software Specification have been mapped into the executable prototype.

**34.5.4** The executable prototypes will normally be partial models of the final SRS and will probably be deficient in areas such as response times. Any areas which cannot be explored during testing of the executable prototype should be recorded, as it is important that the limitations of the executable prototypes are documented. Typically those parts of the Software Specification which are specific to the final implementation, such as built-in test, will be omitted from preliminary validation and explored during the design phase.

**34.5.5** The executable prototypes should have easily-used interfaces to encourage the representatives of the customer and of the Design Authority to use them to explore the functionality of the Software Specification. However, executable prototypes should also be tested, using formally recorded test cases and results. These tests should be designed to meet specific criteria which have been planned in advance. It would generally be inappropriate to use white box test techniques, except where these techniques support a coverage argument for the coverage of the functionality of the formal specification. Black box techniques, such as equivalence partitioning would be appropriate.

**34.5.6** The development, formal verification, testing and limitations of the executable prototypes should be documented in the Specification Record, as should any changes which are made to the Software Specification as a result of the preliminary validation.

**34.5.7** By preserving the Executable Prototype Test Specification for use during validation testing, the Software Design Authority demonstrates continuity of requirements from the beginning to the end of the SRS development process.

**34.5.8** The use of a particular control or data structure in the executable prototypes should not be taken to mean that the final SRS should use the same structure. The control and data structures in the SRS should be determined after careful investigation of the practical alternatives, during the design process.

**34.6** Checking and review of the specification process. No guidance.

## 35 Design Process

**35.1** The objective of the design process is to make a commitment to particular control and data structures. Whereas the Software Specification defines the properties that should be satisfied by the implementation, the Software Design determines how the properties should be achieved. The use of formal methods for design enables the design decisions on control and data structures to be verified by formal arguments.

**35.2** The Software Design normally requires several levels of abstraction, such as architectural design, detailed design etc. Each level should include all the information necessary for the subsequent level. For example, at architectural design, the requirements that each structure will implement should be recorded.

**35.3** The requirement for the Software Design to be constructed in a manner which permits justification that it meets its specification tends to restrict the features and styles that may be used. The Design Team should pay careful attention to the following:

a) Analysability: Some design options will make correctness hard to assure and this will be

**35.3** (Cont)

reflected in proof obligations that are difficult to discharge. The Software Design Authority should balance the design options against the ease of verification, bearing in mind the availability of methods and tools.

b) Complexity and Size: It is usual for the Code of Design Practice to specify limits for the complexity and size of individual modules, because smaller, simpler modules are generally easier to analyse and test and less likely to contain mistakes. Although the overall size and complexity of the SRS will generally be determined by constraints outside the control of the Software Design Authority (ie by the overall system design), the Software Design Authority should ensure that the Software Design does not increase the size and complexity to an unacceptable level. For example, a reconfigurable implementation may be much more complicated, and ultimately less safe, than a simple design which is specific to the application. If necessary, the Software Design Authority should also consider proposing that the Design Authority changes the system design to partition the software into smaller and more dependable parts.

c) Modularity: Comprehension, analysability and testability are all enhanced by appropriate modularity, ie if the SRS is broken down into a hierarchy or network of modules such that each module is small enough to be understood as a whole. However, the interface between modules needs careful design, as too many small modules may lead to a complex data flow that is itself hard to understand. There are tools which will provide metrics indicating high coupling or linkage. Cohesion of modules is harder to measure, but high cohesion will help to reduce linkage. Good modularity is assisted by modern high-level languages and by techniques such as structured design. Since in general smaller modules are much easier to analyse, the size of individual modules and of the overall SRS needs careful consideration.

d) Encapsulation: Encapsulation describes the degree to which all aspects of a particular part of the problem space can be dealt with in one place. Good encapsulation helps to avoid moving complex data flow between many modules. By restricting access to modules to well-defined interfaces, encapsulation also protects data and functions from being accessed or updated in inappropriate ways. Good encapsulation is assisted by languages and methods which enable data and functions to be brought together in modules, and which enforce module boundaries.

e) Abstraction: Abstraction is the degree to which the design methods and tools enable detail to be hidden when an overview of the operations and data of the SRS is required. Abstraction is an essential property of methods and languages used for refining a formal specification. The methods and tools should enable abstraction of both functions and data.

f) Concurrency: Many of the proof theories which are used in the analysis of source code have an implicit assumption that there is no concurrency. Therefore, any decision to use concurrent features, such as tasking, should be balanced against the limitations on methods and tools available for analysis. In addition the extra work involved if a sequential modelling method is adapted to handle a specific concurrent model should be considered. Complex concurrent designs are likely to be hard to analyse and hence to justify. The traditional alternative to concurrent designs involves the use of schedulers. The design of the scheduler should ensure that execution is predictable and that analysis can be applied to show that critical timing requirements will be met. Distributed systems inevitably introduce an element

**35.3** (Cont)

of concurrency and constraints on the behaviour of each node should be set to ensure that the behaviour of the overall system is predictable.

g) Floating point: The use of floating point arithmetic may simplify the source code but introduces additional validation issues for floating point co-processors and for any built-in or library algorithms. The use of floating point or fixed point arithmetic (where the compiler provides scaled integers) may complicate the task of object code verification. For a particular application, the Software Design Authority should balance the difficulty of validating fixed or floating point arithmetic against the additional complexity added to the application code by the use of scaled integers.

h) Recursion: Recursion often provides an elegant mathematical expression of the formal specification or design and is invaluable in proofs. Memory usage is generally difficult to predict when recursion is used. Therefore, recursion should not be used in the implementation unless strict bounds on the depth of recursion can be defined and hence it can be shown that the permitted memory usage will not be exceeded. An exception to this is in the development of compilers where, in general, bounds on memory usage will not be a critical issue.

i) Interrupts: For some applications, the use of interrupts may result in the simplest, most understandable and most reliable design. However, interrupts should be used with care, and justification for how they are used, and why this use is safe should be provided. The contents of registers and shared variables should be considered, as well as when interrupts may or may not occur, and the maximum possible frequency of interrupts. Generally, static analysis techniques do not support the use of interrupts and an application-specific model should be developed. For this to be practical, the use made of interrupts should be simple and well understood.

j) Dynamic memory allocation (heaps): The use of pointers is a natural way to declare certain data structures, including linked lists, graphs and trees. It is almost inconceivable to imagine developing a compiler without the use of pointers. Pointers may reference global areas where objects are allocated and their storage is managed (the heap space). The main concern with using dynamic memory allocation is the difficulty of predicting whether the memory space allowed is adequate, particularly if it becomes fragmented as it is reused. The use of dynamic memory allocation is not a problem provided that exhausting the memory space does not constitute a dangerous failure. Note that even if the application code does not use dynamic memory allocation, the compiler may well use memory space, typically when arrays and records are passed as parameters to procedures and functions. Object orientated languages generally require the use of dynamic memory allocation. For most real time systems, heap space should only be used when:

i) the operations are deterministic;

ii) the pointers are typed (ie the space required for an object can be predicted);

iii) the space used is bounded;

iv) storage fragmentation does not occur.

**35.4** To ensure consistency throughout the Software Design, and an efficient design process, the Software Design Authority should establish the general principles and style for the design before working on the detail. Non-functional requirements tend to manifest themselves as properties rather than as functions of the final SRS and it is important that a considered approach is made to the following:

a) Defensive code: Defensive code has the disadvantage of adding code to the application and of making it more difficult and potentially harder to test. Against this should be weighed the increase in safety integrity gained by preventing errors from propagating and by having the ability to detect an unexpected error and return the SRS to a safe state. Before any defensive code is written, the behaviour of the compiler when it detects statically determinable constructs should be well understood, as the compiler may optimise out the defensive code. The option to include or exclude compiler generated checks should be considered, as should the likely consequences of a compiler generated check detecting an error. Defensive code may be efficiently implemented using `exception' features of the language if these are included within the language subset. Defensive code should follow a consistent plan, and should be identified as such by source code comments. In general, consideration should be given to adding defensive code to:

i) protect the software against the hardware (for example range check inputs to the SRS and self-test routines such as RAM tests);

ii) protect the hardware against the software (for example range checks outputs even when they can be statically determined always to be within range);

iii) protect the software against the software (for example range check inputs and outputs for each procedure, add `others' clauses to `CASE' structures, explicitly check assumptions and defend against domain incorrect errors such as divide by zero).

b) Fault-tolerance: Fault-tolerance may be necessary where high availability is required, but it should be used with care. Like defensive code, it complicates the SRS and may make analysis difficult. Fault tolerance should be founded on statically determinable criteria (such as two parallel, equally justified SRS systems) rather than on dynamic features such as dynamic recovery blocks. A case may be made for recovery blocks, provided that the behaviour of the code is determinable at all times. Behaviour under both persistent and intermittent fault condition should be considered. `Fail safe' measures, such as graceful degradation or placing the software into a safe inoperable state should be considered, especially where the software detects that its own behaviour or its inputs are not within the designed parameters.

c) Reliability: Reliability of software relates only to design faults (correctness), as software does not wear out (unlike hardware). Strictly, the correctness of software is either 0 or 1, since errors are either present or not. From a practical point of view, quantifying software reliability is therefore a difficult issue and is concerned not just with the presence of errors, but with the circumstances in which the errors are revealed, and the consequences of the errors. The only means by which achieved reliability can be measured directly is based upon statistical testing (such as that suggested under validation testing). High reliability in software should be achieved by the diligent application of the principles in this Standard. It is important, however, that the Design Team appreciate that there may be a difference between

**35.4**(Cont)

safety and reliability: a system with an unreliable `on' switch, that prevents it operating may well be totally safe, although completely unreliable. For reliability, the Design Team should consider both availability (ie the probability of the system generating outputs on demand; starting up; running without crashing and running without halting due to an error) and reliability of the output (ie the likelihood of an incorrect, inaccurate or inappropriately timed output).

d) Accuracy: The Design Team should assess the ability of the algorithms to produce outputs which are both stable and meet the accuracy requirements of the SRS. If, as previously discussed, floating point arithmetic is deemed inappropriate, the Design Team will be left with the option of fixed point or scaled integer, or more complex solutions such as modular arithmetic. Whichever solution is adopted, the Design Team or V&V Team should undertake a computational methods study to ensure that the outputs are within the specified accuracy, that discrete algorithms model continuous, real behaviour with sufficient accuracy and that overflow (or underflow) does not occur.

e) Capacity: When the amount of input or data that can be processed is variable, for example because of interrupts, or the requirement to store and process records, the Design Team should ensure that they can predict the maximum demands that could be made upon the system. The system should be shown to be robust and operable at the maximum demands, and stress tests of maximum capacity should be included during the testing phase. The Design Team should also ensure that the target has the required capacity for the SRS and should check:

i) stack sizes;

ii) buffer and queue sizes;

iii) static memory allocation;

iv) dynamic memory (heap) allocation;

v) temporary file or other memory usage.

f) User-interface: Where the SRS has an interface directly to a user or operator, extra care should be taken in its specification and design. A prototype should be constructed and the end user should be involved in ensuring that the SRS will function satisfactorily under operational conditions. Where the safety of the system could be affected by the actions of the user as a result of output from the SRS or via input to the SRS, human factors experts should be involved in designing and checking the user-interface and testing the usability of the final system.

g) Software configuration and version identity: All parts of the SRS should be identifiable and should have a recorded version identity. It should be possible, via the configuration management system, to relate the version of each part to the versions of all other parts to which it relates. The Design Team should consider adding explicitly embedded identifiers, including a version identifier, into the final version of the object code. Where configurable or

**35.4** (Cont)

programmable units are used in a larger system, the use of a start up or self test procedure which checks the identity of the parts of the SRS should be considered. Behaviour in the event that incompatible versions are detected should be defined.

**35.5** Verification of the Software Design

**35.5.1** For each stage of the design, syntax and type checking of the formal parts should be carried out. Formal verification of consistency and of refinement of both functionality and data from the Software Specification or previous stage should be performed, by constructing and discharging all appropriate proof obligations.

**35.5.2** Non-functional requirements should be checked by conducting performance modelling (for example by the analysis of timing and accuracy) on the Software Design, to ensure that the relevant requirements will be met even in the worst case. The analysis may require that prototypes whose actual performance can be measured are developed, so that the measured data can be used in analysis. Performance modelling should be conducted as early as possible in the design, even if only rough estimates are available. If performance constraints cannot be met this may result in significant changes to the design, which would otherwise occur at a very late stage.

**35.6** Checking and review of the design process

**35.6.1** Design reviews should be conducted at regular intervals, and the amount of material reviewed should be consistent with each reviewer needing only a few hours preparation with the review itself lasting for an hour or less. This inevitably means that the Software Design will be reviewed in parts. Periodically therefore, the reviewed parts should be subject to an overall review which considers the wider issues and which looks at the Software Design as a whole.

**35.6.2** Performance modelling should also be subject to review by the V&V Team as significant combinations of circumstances which would result in a failure to meet the performance constraints may otherwise be overlooked.

**36** Coding Process

**36.1** Coding standards

**36.1.1** The coding standard should:

a) restrict the use of language features which are not statically determinable or where the implementation is compiler dependent;

b) keep modules small, while minimizing data flow between modules;

c) discourage the use of global variables and data;

d) encourage reuse as far as this assists comprehensibility, maintainability and integrity;

**36.1.1** (Cont)

e) enhance readability by the appropriate use of upper and lower case letters, meaningful identifiers, comments, blank lines and indentation;

f) maximize use of the strong typing facilities provided by the language;

g) avoid use of complex or obscure language features which programmers may have difficulty in using;

h) enforce the use of a simplified language syntax, using as far as practical one syntactical structure at a time (for example nesting constructs within constructs, such as function calls as parameters to procedures, should be avoided).

**36.1.2** When low level or assembly languages are used:

a) the total amount of such languages should be kept very small;

b) a modular coding style should be used with each module being kept small;

c) the coding standards should enforce a style which mimics the control flow constructs of high-level languages (for example avoiding jumps except to create structures such as subprogram calls, if..then..else, case, and single entry, single exit loops);

d) the density of comments should be such as to render the code understandable;

e) language features which permit named identifiers, rather than addresses, should be used;

f) the coding style should be such as to make it easy for type checking to be performed.

**36.2** No guidance.

**36.3** The Code of Design Practice should contain procedures and guidance on how to avoid or detect known faults in the compilation system and target hardware.

**36.4** No guidance.

**36.5** Static analysis and formal verification

**36.5.1** Whether the whole or part of the Software Specification and Software Design have been written in a formal notation, static analysis should be performed on the whole of the SRS. If the whole Software Specification and Software Design have been formally specified and designed, and all relevant proof obligations have been constructed and discharged, then the static analysis will only be required to demonstrate correct usage of the implementation language. Where only part of the SRS has been treated formally (because only part is relevant to safety), static analysis should be used to show that there are no faults in the non-formally specified parts which might affect the formally specified parts (such as writing to an array with an out-of-range index). Note that the requirement for the code to be free of anomalies that would affect the safety of the system will generally require the use of all the static analysis techniques listed in **26.2.4**.

**36.5.2** The requirements of this Standard for static analysis should be satisfiable with the help of a suitable tool, even if the process is only semi-automatic (ie the tool provides assistance in the execution of a task, rather than performing it totally automatically). Manual static analysis should not be used.

**36.5.3** The requirement to construct and discharge proof obligations which cover the final refinement step from Software Design to code is often performed using static analysis tools.

**36.6** Object code verification

**36.6.1** It is unlikely that compilers for wide scope, high-level languages will be produced which can be formally proven to produce correct code, and there is therefore a possibility that the compilation system may introduce an error into the SRS that affects the safety of the system. The Software Design Authority should demonstrate by formal, static methods or by a justified argument that such an event will not occur.

**36.6.2** If formal, static methods are used, the object code should be treated as the implementation of the source code, with demonstration of its correctness being achieved either by construction and discharge of refinement proofs, or by static analysis. This analysis should be concerned only with showing that the object code is a correct implementation of the source code. Correctness of the source code should have been demonstrated via the provisions of **36.5**. Listings generated by the compiler may be used as a guide in the analysis, but they should not be assumed to be correct. Memory assignment can often be addressed by performing the analysis left to right (ie from initialisation) in addition to bottom up (as is usual for static analysis). Left to right analysis enables memory addresses to be matched with the identifiers in the source code.

**36.6.3** If the static approach is inappropriate a justification should be produced which is based partly on testing the object code, partly on additional analysis which is performed and partly on the known performance of the compilation system (see **28.6**). Whichever approach is used, the ease of object code verification is heavily dependent on the combination of implementation language, compilation system and coding standard. Embedded, real-time software tends to require language features which are difficult to test via generic compiler validation and testing.

**36.6.4** Where good commercial compilers of well designed, standardized languages are used, the likelihood of errors being introduced by the compilation system is probably much less than that of errors existing in the design or source code. In these circumstances, the rigour of justification required for the final implementation step, from source code to object code, is less than that required from the specification to the design or from the design to the source code. This Standard permits this justification to be made using a single argument, relying on a combination of analysis and testing. Except for trivial systems, testing can never be exhaustive but it does have the advantage of giving some confidence in the hardware on which the software is dependent. Justification is likely to be more difficult for an immature language and compilation system, than for one which is equally well designed and has been extensively used. Experience shows that extensive use of a language, resulting in identification of language insecurities, leads to more robust compilers.

**36.6.5** Object code verification based on testing might be achieved by the procedures described in **37.3**, but it is likely that, even if the same tests are used, special arrangements will need to be made to show that the object code has been verified. Tests should be conducted using the final version of the object code. If the language supports libraries and separate compilation, then it is should be possible to test the object code in parts and justify the linking separately. Test coverage should be shown using some form of hardware monitor, since coverage should be shown at machine instruction level the final version of the object code is not the instrumented code. To achieve object code coverage, it will probably be necessary to restrict the ability of the compiler to select the evaluation order of `ands and `ors in a single branch. Branch coverage is probably impractical at this level. The object code verification essentially relies on the much tougher targets imposed by 38.2.2 of Part 1 of this Standard for testing the source code. For this reason, it is necessary that the mapping between the source and object code is well understood and documented. The ease with which this can be done is language and compiler dependent.

**36.6.6** Lists of known compiler bugs should be used to determine which language features are likely to be error prone.

**36.6.7** If object code verification is performed by testing, the logical place to record the activity is in the Test Record. If however, the justification of the object code is shown by formal correctness proofs of the compilation process, then these will more logically fit in the Design Record.

## **36.7** Checking and review of the coding process

**36.7.1** The static analysis should be reviewed, paying especial care to any anomalies found, to ensure that adequate and correct explanation of all anomalies has been recorded, and genuine errors have not been explained away.

**36.7.2** The results of the static analysis and formal verification should be used in reviewing the source code.

**36.7.3** The reviewers should assess the stated safety assurance of the compilation system and determine whether adequate object code verification has been performed. The object code verification should be reviewed, remembering that the purpose of object code verification is to check the output of the compiler and linker, against the source code; not to check the correctness of the Software Design.

**36.7.4** If the object code verification is recorded in the Test Record, then the review should also be recorded in the Test Record.

# **37** Testing and Integration

## **37.1** Principles of testing

**37.1.1** Where full formal verification of the Software Specification and Software Design has not been performed, the amount of testing will need to be increased. Sufficient testing should be performed to show that the dynamic and performance requirements have been met, and that the assumptions used by the formal methods are valid for the target system.

**37.1.2** Many of the criteria for test selection (such as boundary value tests) result in the production of a set of test cases which are unrepresentative of the operating conditions of the SRS. Validation testing should correct this imbalance. The Software Safety Case requires validation testing to act as a diverse argument from the one of formal demonstration of correctness.

**37.1.3** Testing should be well planned, and should be documented, reviewed and justified in advance of executing the tests. Objective evidence of the extent of testing should be monitored by means of both white box (eg statement, branch, Boolean) and black box (eg requirement, equivalence partitions) coverage metrics.

**37.1.4** The testing phase provides an opportunity to introduce additional independence into the SRS development process. Tests may either be designed by the V&V Team, or designed by the Design Team and reviewed by the V&V Team.

**37.1.5** It should be possible to design the tests without consulting the code. Such tests provide the most rigorous exercise of the code. If it is found that designing the tests in this way is not practical, the Software Design should be reviewed and improved. If after executing the tests, it is found that the required coverage has not been achieved, extra tests should be added once it has been shown that the code does not implement additional functionality not specified in the Software Design. If test design is automated, the automated test design tools should use the Software Specification and Design, and preferably the formal representation, rather than the code.

**37.1.6** Once code has been amended in any way, any tests which have already been run should be rerun. This is generally known as regression testing.

**37.1.7** This Standard requires repeatability of testing. This is only achievable if all elements listed, any one of which may affect the outputs of a test, have been identified and placed under configuration control. Tests which are driven by user input or which depend on real time events should be given special consideration in the test planning stage. Using a test tool which simulates the user input and captures and analyses screen output will enhance repeatability. Repeatability may be achieved by controlling the environment such that the expected result will lie between tolerance bands. Test harnesses can also be designed to force a deterministic, repeatable scenario.

**37.1.8** The expected test results should be documented in such a way as to allow a comparison with the actual outputs to be made. In many cases, the comparison may be made in an automated fashion. There may be cases where there are differences between the actual and expected outputs which do not indicate a test failure, because the differences are within the specified tolerances of the SRS. An explanation should be provided for all such discrepancies.

**37.1.9** The most valuable form of testing is that conducted on the real target hardware using the real code, and this is the form of testing which provides most weight in arguments on the safety of the SRS. It may, however, be desirable to pre-run the tests on a host platform (either to debug the tests or to provide better analysis of the test coverage). If the real target hardware cannot be used for the testing, hardware emulators can be used, but justification should be provided for the correctness of these emulators.

**37.1.10** For testing purposes, it is sometimes necessary to instrument or add code to the code under test. Instrumentation takes many forms, including adding source code, adding object code, changing compiler switches to remove optimisation or invoke a debugger, or changing the mode of the microprocessor to execute additional microcode. Where this is necessary, consideration should be given to the possibility of running all tests twice; once instrumented, and once uninstrumented. Otherwise, a justification should be made for the test instrumentation to show that its inclusion does not affect the integrity of the results. This would indicate that although there is instrumentation to monitor coverage, drive the test or capture the results, the object code of the module under test is identical to the operational software.

**37.2** No guidance.

**37.3** Unit, integration and system testing

**37.3.1** The goal of unit testing is to demonstrate that each unit performs as specified on all paths. Since this is impractical in many cases, the demonstration should use some limited form of path coverage, such as Linear Code Sequence and Jumps (LCSAJ), or paths limited to single iterations of loops. Where formal verification has taken place before testing, it is likely that many of the errors that would otherwise have been detected by testing will have been removed. It may therefore be possible to justify that unit and integration testing be reduced where full formal development of the SRS has taken place. Automated test design, using the formal representation of the Software Specification and Software Design should be considered as a method of generating sufficient tests to achieve the required coverage. The targets for test design stated in this Standard are for a minimal coverage set, but are not intended to be combinatorial (ie it is unlikely to be necessary, or even possible, to set all variables to minimum and maximum values on every branch).

**37.3.2** Many tools provide a variety of other coverage metrics which encourage systematic testing and which provide better indication of the extent of the coverage of the SRS code. Justification should be provided for the coverage metrics chosen. Targets for each metric should be set in the Software Verification and Validation Plan. The best metrics to choose are those for which the targets can be set to 100% (at least of semantically feasible combinations), since where a target is less than 100%, the combinations tested are likely to be the simplest ones. Justification for the selection of targets should also be provided.

**37.3.3** Experienced test designers should use engineering judgement to select additional test cases which are likely to discover errors. All test values should be set explicitly on each test to ensure that tests remain self-consistent if extra tests are added. To achieve greater coverage of the values that each variable can take, variables may be given different values on each test. To ensure that variables are not confused with one another, consider ensuring that values of the variables are different from each other. Variables which can approach or take the value 0 should be given special consideration, as problems with accuracy and instability may become more apparent with near 0 values. Other examples to consider are 1, -1, 90, etc. Test variables should be exercised at the boundary conditions to ensure that final and intermediate values do not result in underflow or overflow conditions.

**37.3.4** For some systems, it may be possible to omit integration testing as a separate exercise, provided that the software interfaces can be shown to be adequately exercised by the system tests.

**37.3.5** System tests should be designed from the Software Specification and the Software Requirement. Traceability via traceability matrices or a traceability tool should be provided between the Software Requirement, the Software Specification and the tests..

**37.3.6** The functions of the SRS should be demonstrated in cases where the SRS will be most likely to fail, for example, with rapidly changing inputs, very large input files or a high numbers of interrupts. The Design Authority may need to provide special facilities for these demonstrations..

**37.3.7** he SRS should be run on the final hardware, with tests designed to show that communication protocols, timing, initialisation of input and output devices etc are correct.

**37.4** Validation tests

**37.4.1** For a statistical indication of the achieved safety integrity of the SRS, the amount of validation testing performed should be consistent with the required safety integrity level. A fully automated test environment should be developed to ensure that large numbers of tests can be generated, executed and checked. Test tools should simulate inputs and capture and check outputs. For small systems running on standard hardware platforms, it may be possible to set up multiple test units to increase confidence in the integrity but this may not always be practical, particularly with large or distributed systems. Large scale validation testing should still be performed, but certification of such systems will depend on engineering judgement.

**37.4.2** Validation testing provides a valid argument for the Software Safety Case if there is independence of the testing team from the Design Team. To eliminate the concern that code may be designed to pass specific tests, non-disclosure of validation tests is required. However, good communications between the Design Team and the V&V Team should be encouraged. The V&V Team should be involved in the verification of the SRS at all stages of the SRS development process and should provide constructive advice wherever it is needed.

**37.4.3** To avoid a potential single point of failure, the validation tests should be designed from the Software Requirement, rather than from the Software Specification.

**37.4.4** The Executable Prototype Test Specification should have been designed or examined by the end users of the SRS. Validation tests should also be developed from this specification and run on the SRS. This closes the loop between preliminary validation and the final implementation and ensures that misconceptions have not developed during the project.

**37.4.5** Testing should pay particular attention to safety, as well as to non-functional and performance aspects of the SRS. Experienced system engineers should be used to develop test scenarios which are most likely to detect problems with the SRS.

**37.4.6** For a statistical argument about the achieved safety integrity of the SRS, the validation testing should use randomized, real or simulated data for inputs. The data should be equivalent to the inputs that the SRS will encounter during its operational life.

**37.4.7** Discrepancies between the system which undergoes validation testing and the final system may potentially undermine the diverse argument in the Software Safety Case. Therefore the tested system should be, as far as possible, identical to the final system, and any differences should be considered and justified.

**37.4.8** Ideally, the validation testing should find no errors at all in the SRS, as these should have been eradicated by the verification performed earlier in the SRS development process. Any errors found therefore indicate a failure in the SRS development process. The possibility of there being other such failures should be investigated.

**37.4.9** As with system testing, it is possible for the actual and expected outputs to differ, due to slight inaccuracies in the outputs that are within specified tolerances.

Section Six. Certification and In-service use

**38** Certification

**38.1** The Design Authority has overall responsibility for the safety of the system within which the SRS is contained. Therefore the Design Authority's Certificate of Design for the system will incorporate (either explicitly or implicitly) certification of the SRS. Where the Design Authority is also the Software Design Authority, the SRS Certificate of Design should be produced in the same way as if the software were developed by a subcontractor, ie retained by the Design Authority and referenced in the supporting documentation to the Certificate of Design for the system.

Certification is the formal process of recording that, as far as the Software Design Authority is aware, the SRS is of adequate safety integrity, the work has been carried out in accordance with this Standard and the SRS meets the Software Requirement. Where a subcontractor to the Software Design Authority has developed the software, the subcontractor should certify their own contribution by signing the SRS Certificate of Design as the software developer.

**38.2** Evidence of adequate safety integrity is provided by demonstrating compliance of the SRS to the Software Requirement, since the Software Requirement contains the safety requirements for the software.

**38.3** No guidance.

**38.4** No guidance.

**38.5** **Annex C** provides a proforma for the SRS Certificate of Design. Where this certificate is not considered to be suitable, the Software Design Authority may submit an alternative certificate, the form of which should be agreed by the Design Authority, the Independent Safety Auditor and the MOD PM. As a minimum, the SRS Certificate of Design should contain:

a) a certificate identity number;

b) identification of the SRS version, mark, model or applicable identifier to which the certificate refers;

c) certification of conformance to this Standard, possibly with exceptions as defined in the Software Quality Plan;

d) certification of conformance to the Software Requirement;

e) reference to the documentary evidence of conformance;

f) identification of the parent system/subsystem (equipment within which the SRS resides) version, mark, model or applicable identifier to which the certificate refers;

g) reference to any restrictions or limitations on use;

**38.5** (Cont)

h) signatures from responsible personnel, as appropriate:

i) the Software Design Authority;

ii) the Software Developer;

iii) the Software Project Safety Engineer (if different);

iv) the Independent Safety Auditor.

**39** Acceptance

**39.1** Acceptance testing by the MOD may not be required for the SRS in isolation if acceptance testing is performed at the system level. Acceptance testing may also include a period of field trials. The Design Authority should ensure that the criteria for acceptance of the SRS from the Software Design Authority are equivalent to the MOD SRS acceptance criteria.

**39.2** The Acceptance Test Specification should be produced by the Software Design Authority at an early stage of the SRS development process. Many of the tests may also form part of the validation process.

**39.3** The results of the acceptance tests should be reviewed by the Software Design Authority, the Design Authority and the MOD PM and any defects discovered should be treated in the same manner as defects discovered during the validation process as discussed in **37.4**

**39.4** No guidance.

**40** Replication

**40.1** A necessary pre-requisite of preserving safety integrity is ensuring that each binary image is correct.

**40.2** No guidance.

**40.3** Replicated binary images should be identical, although it is permitted for the value of an embedded identification number to differ between copies, provided that the different numbers occupy the same location and the same amount of memory.

**40.4** Checks should be performed in order to gain assurance that an executable binary image has been replicated without change. These will typically involve the use of checksums and reading the replicated image for direct comparison against the original master copy.

**41** User Instruction

**41.1** The User Manuals should identify the different classes of user for whom they are intended, for example operators, installers or end users. Separate User Manuals may be provided for each different class of user. The User Manuals for the software may be

**41.1** (Cont)

encompassed by the User Manuals for the system. The quality of User manuals produced is important and it is recommended that BS7649 (Guide to the Design and Preparation of Documentation for Users of Applications Software) or SRDA-r1 "Developing best Operating Procedures - a Guide to designing Good Manuals and Job-aids", ISBN 085-3563594, be consulted.

**41.2** The interface with the user is a potential source of hazard on complex software-driven systems. In particular, user mistakes can occur if the software operates in a way that the user does not anticipate. Logic combinations in even relatively small software driven systems can be enormous, but it is important the user is aware of the principles of the logical design, and of any logic conditions that may have a safety implication, so as to minimize any chance of the user mismanaging the system/software. It is therefore vital to ensure that the appropriate software-related instruction is promulgated to the end user, especially when the user only has access to the system level User Manuals.

**42** In-service

**42.1** The Software Design Authority for the in-service phase may not be the same as that responsible for the original development of the software, and may or may not be the same as the in-service Design Authority (ie the Design Authority responsible for the whole system for the in-service phase).

**42.2** The anomaly reporting procedures for the SRS should be part of the Data Reporting Analysis and Corrective Action System (DRACAS) that the in-service Design Authority is required (in Def Stan 00-56) to set up and maintain.

**42.3** The in-service Design Authority should ensure that all in-service anomalies reported under the DRACAS are transmitted to the in-service Software Design Authority so that the in-service Software Design Authority can determine any relevance to the safety of the operation of the software.

**42.4** No guidance.

**42.5** No guidance.

**42.6** Each anomaly report should:

a) give the date of the anomaly and of the report;

b) identify the personnel concerned;

c) identify the configuration of the system and the version number of the SRS;

d) state the problem and the conditions that produced it, including whether the system was being operated within its design envelope;

e) assess the actual or likely consequences;

f) describe the corrective action taken.

**42.7** Although the principles for maintaining SRS are much the same as those for the maintenance of other software, the problems to be addressed are likely to be far more demanding. When planning the maintenance of SRS, consideration should be given to:

a) the problems involved with large scale revalidation of small scale changes;

b) which configurable items will be held in an operational state and which will be re-activated when required;

c) the expected life of the hardware on which the SRS is installed;

d) the availability of the support tools.

**42.8** The content of the Software Maintenance Plan should be as detailed in **annex B.22**.

Section Seven. Application of this Standard Across Differing Safety Integrity Levels.

**43** Software of differing Safety Integrity Levels

**43.1** Part 1 of this Standard applies in its entirety to Safety Critical Software - ie Software of Safety Integrity Level S4. For software of lower safety Integrity levels (S1 to S3) a less rigorous approach to the interpretation of some clauses may be acceptable, subject to justification.

**43.2** **Annex D** provides a tailoring guide for levels S1 to S3 with respect to clauses where justified interpretation may be acceptable. The degree of compliance to this Standard is indicated by the levels `M', `J1' and `J2' for the varying Safety Integrity Levels specified.

**44.3** No guidance.

Collation page

Index

Bibliography

The references given in this annex are from the open literature. Entry in the bibliography implies no bias towards any particular product for use by the MOD. Omission from the bibliography implies no bias against any particular product that could be demonstrated to meet MOD requirements. Exceptionally, the MOD may declare a preference, for example in the use of Ada.

**A.1** IEC 1508 Functional Safety: Safety Related Systems.

**A.2** JPS Joint MOD/Industry Policy for Military Operational Systems

**A.3** B A Carré and T J Jennings. SPARK - The SPARK Ada Kernel. University of Southampton, March 1988.

**A.4** Cullyer, S J Goodenough and B A Wichmann, "The Choice of Computer Languages in Safety Critical Systems", Software Engineering Journal, Vol 6, No 2, pp51-58, March 1991.

**A.5** T Gilb and D Graham. Software Inspection. Addison-Wesley 1993. ISBN 0-201-63181-4.

**A.6** J Rushby. Formal Methods and the Certification of Critical Systems. SRI International. 1993. SRI-CSL-93-07.

**A.7** B A Wichmann, A A Canning, D L Clutterbuck, L A Winsborrow, N J Ward and D W R Marsh. An Industrial Perspective on Static Analysis. Software Engineering Journal. March 1995, pp69-75.

**A.8** B A Wichmann. Insecurities in the Ada programming language. NPL Report DITC 137/89, January 1989.

**A.9** B Littlewood and L Stringini. Assessment of ultrahigh dependability for Software-based systems. CACM, Vol 36, No 11, pp69-80, 1993.

**A.10** The British Computer Society. Industry Structure Model, Release 2, 1991.

**A.11** Software Engineering Institute (SEI). Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-24, Feb 93.

**A.12** P Bishop et al, The SHIP Safety Case - A Combination of System and Software methods. 12th Annual CSR Workshop, September 1995.

**A.13** J A McDermid, Support for Safety Cases and Safety argument using SAM. Reliability Engineering and Safety Systems, Vol 43, No 2, pp111-127, 1994.

**A.14** M A Ould, Business Processes - Modelling and Analysis for Re-engineering and Improvement. Wiley, 1995.

**A.15** ICAM Architecture Part II - Vol IV - Functional Modelling Manual (IDEF0), AFWAL-TR-81-4023, Materials Laboratory. Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio 45433, June 1981.

**A.16** I J Sinclair, The Use of Commercial Off-The-Shelf (COTS) Software in Safety-Related Applications. HSE Contract Research Report No 80/1995, ISBN 0-7176-0984-7.

**A.17** S M Austin, D R Wilkins and B A Wichmann. An Ada Program Test Generator. TriAdaa Conference Proceedings. ACM, October 1991.

**A.18** J Garnsworthy et al. Automatic Proof of the Absence of Run-Time Errors, Ada UK Conference, 1993.

**A.19** N E Fenton. Software Metrics A Rigorous Approach. Chapman and Hall, 1991.

Documentation

This annex lists the software and documents to be produced in accordance with Part 1 of this Standard.

As stated in annex B to Part 1, the Software Design Authority may decide, in conjunction with the Design Authority and the MOD, that the information would be better presented using a different contents list to that presented here. For example a number of documents may be incorporated into a single document, provided, firstly, that the complete document set incorporates all the contents listed in this annex and, secondly, that there is easy traceability between these requirements and their incorporation into the actual document set.

## **B.1** Software Safety Plan

The Software Safety Plan for the production of SRS contains:

a) A definition of the purpose and scope of the Software Safety Plan, including those parts of the Software Safety Case that are expected to be achieved by adherence to the plan.

b) Definitions and references.

c) Details of the management of the production of SRS, including:

i) the identification of the organizations and key personnel required by this Standard, including Design Authority, Software Design Authority, Project Manager, Software Project Manager, Project Safety Engineer, Design Team, V&V Team, Independent Safety Auditor, relevant subcontractors or suppliers;

ii) CVs of key individuals who will be involved with the production of the SRS including the leader of the Design Team, the V&V Team and the Independent Safety Auditor;

iii) a justification that the seniority, authority, qualifications, training and experience of all staff directly involved in the management, development, verification, validation and production of the software are appropriate for their assigned tasks;

iv) a description of the interfaces between the MOD PM, the Independent Safety Auditor, the Design Authority, Software Design Authority and their respective subcontractors;

v) details of adequate resource planning, including, but not limited to, finance, personnel, equipment and tools;

vi) a list and table of contents of the deliverable items to be produced;

vii) details of the certification process;

**B.1** (Cont)

viii) the definition of the circumstances under which matters concerning safety should be referred between the Design Authority, the Software Design Authority or its subcontractors, the Independent Safety Auditor and the MOD PM;

d) specification of the criteria for tool approval and limitations on the use of certain tools;

e) the data to be collected for analysis purposes;

f) the procedures for evaluating the safety integrity of previously developed or purchased systems;

g) details of the way in which the Software Safety Case is to be updated throughout the project lifecycle as the plan is applied and evidence is accumulated.

h) reference to the Software Configuration Management Plan, Code of Design Practice and Software Development Plan.

i) details and scheduling of the software safety reviews;

j) details and scheduling of the safety documentation.

**B.2** Software Safety Case.

The Software Safety Case presents a readable justification that the software is safe in its specified context, and includes:

a) A summary of the equipment-level design approach to safety, addressing:

i) a summary of the equipment safety case including a brief description of the system functions and safety features and their means of implementation by software, hardware or system architecture;

ii) the safety integrity levels of the software and hardware components

iii) the assignment of the software and hardware components to segregation units;

iv) a description of the processors and other hardware devices with an interface to the SRS.

b) A description of the role of the SRS in ensuring safety, with a summary of the software safety properties. The software safety properties should be related to the safety integrity levels defined in Def Stan 00-56 and any standards mandated at the equipment level. Derived safety requirements that emerge during equipment design should also be included.

c) A description of the software architecture and the way in which it is appropriate for the safety integrity requirements for the SRS, including;

**B.2** (Cont)

i) the means used to segregate functions of different criticalities and to prevent failures propagating;

ii) a statement of the software components developed;

iii) a statement of the object code size, timing and memory margins, resource limitations, and the means of measuring each of these characteristics.

d) A description and justification of the two or more diverse arguments used to demonstrate that the software satisfies its safety requirements, including:

i) Analytical arguments, including:

formal arguments that the object code satisfies the formal specification;
analysis of performance, timing and resource usage;

analysis of the effectiveness of fault detection, fault tolerance and fail safety features;

formal arguments used in preliminary validation to show that the formal specification complies with the safety requirements in the Software Requirement.

ii) Arguments from testing, including validation testing of the SRS.

e) A description and justification of the SRS development process, including:

i) An analysis of the Code of Design Practice and the Software Safety Plan and an assessment of their suitability for the safety integrity level of the SRS, including:

a description of the faults and hazards that will be specifically addressed and minimized by the SRS development process;

an appraisal of the competence of the staff involved in the project.

ii) An analysis of historical data on the safety integrity of software developed using the proposed or similar development processes.

iii) A safety analysis of the methods, tools and support software employed, justifying their use with respect to the effects of possible faults in them on the safety integrity of the SRS.

iv) A justification for the inclusion of any commercial off-the-shelf software (COTS) or other previously developed software.

f) A description of any outstanding issues that may affect the safety integrity of the SRS, and a statement of progress with respect to the Software Safety Plan and Software Development Plan.

**B.2** (Cont)

g) A summary of changes to the Software Safety Case with a statement of the reason for the change and an analysis of the significance and implications for safety of the SRS.

h) An analysis of the compliance with this Standard, the Code of Design Practice, the Software Quality Plan and any other standards or guidelines referenced in the Software Safety Plan. A copy of the SRS Certificate of Design should also be included for the operational Software Safety Case as should a statement of any concessions that have been negotiated with the MOD PM.

i) A description of any field experience with the SRS or any part of it, including an analysis of the impact of any software failures on the safety of the equipment, addressing the potential consequences of the failure, its root causes, the relevance to the present application, and the lessons for the software development process. The in-service feedback should include an analysis of the experience from any variants of the equipment.

j) The software configuration and versions of hardware, tools and support software that this Software Safety Case refers to, defined by part number and version as set out in the Software Configuration Management Plan.

**B.3** Software Safety Records Log

The Software Safety Records Log contains or references:

a) All the data that supports the Software Safety Case.
It provides:

i) the evidence that the Software Safety Plan has been followed and that the results of following that plan have been satisfactory;

ii) the results of all V&V activities;

iii) the results of software safety reviews;

iv) the results of the safety analysis of the SRS;

v) the results of software safety audits.

b) Details of supporting analyses for the Software Safety Case including:

i) an analysis and assessment of the Code of Design Practice and Software Safety Plan;

ii) an analysis of historic software failure data and software development process data for the system being developed and other similar systems;

iii) an analysis of the SRS development process including an analysis of the adequacy of the methods and tools used.

**B.4** Software Safety Audit Plan.

The plan for the activities of the Independent Safety Auditor, containing:

a) a definition of the purpose and scope of the Software Safety Audit Plan;

b) details of the management structure and control for the audit programme;

c) details of planned updates to the Software Safety Audit Plan;

d) a description of the activities to be carried out, including their scope and depth;

e) details of the frequency and timing of auditing activities;

f) a description of how the audit programme is to be integrated with the safety programme, including a description of the interactions and dependencies of the audit programme with the Software Development Plan, the Software Verification and Validation Plan and the Software Safety Plan.

g) a specification of the contents of the Software Safety Audit Report.

**B.5** Software Safety Audit Report.

The Independent Safety Auditor's report. A number of reports with varying contents are likely to be produced but, as a whole, the Software Safety Audit Report includes:

a) a brief description of the SRS and of its function;

b) a definition of the scope of the audit and a description of the work done in carrying it out;

c) a review of progress against the Software Safety Audit Plan;

d) an assessment of the veracity and completeness of the Software Safety Case;

e) an assessment of the adequacy of the Software Safety Plan;

f) the results of audits of the project against the Software Safety Plan with respect to the evidence accumulated in the Software Safety Records Log;

g) an assessment of compliance with the requirements of this Standard;

h) references to all documents examined;

i) conclusions and recommendations.

**B.6** Software Quality Plan.

The plan for application of the quality assurance activities to the SRS. The plan includes the following, which may contain additional detail to the requirements of other standards that are also to be applied on the particular SRS development:

a) definition of applicable standards, procedures and plans, including this Standard, applicable company QA procedures and other plans governing the SRS development;

b) clear identification and details of any allowable variations from this Standard and from company procedures;

c) description of the Software Design Authority organization and responsibilities relating to software quality assurance activities on the SRS development;

d) details of the quality assurance and audit activities relevant to the SRS development, including information on when they are to be applied;

e) definition of problem reporting and corrective action practices and responsibilities;

f) definition of the review procedure for documentation.

**B.7** Software Development Plan

The plan for the development process and its management. The plan includes:

a) definition of the SRS development project objectives;

b) details of the project organization, including management and reporting structure, responsibilities and internal and external interfaces;

c) overview of the SRS development;

d) definition of the outputs from the SRS development process, including documents and the phase at which each is produced;

e) definition of phase transition criteria;

f) project management and control information, including schedules, plans, budgets, resources, staffing plan, milestones, project risk management, monitoring/measuring activities and details of the software integration planning;

g) details of the metrics to be recorded, definition of target or allowable values, and explanation of how the metrics are to be interpreted as a measure of development process quality (may alternatively be included in a stand-alone Metrics Plan);

h) details of the software development environment, including tools, methods and techniques, programming languages and development platform, with reference as appropriate to the Code of Design Practice and Software Safety Plan

**B.8** Software Risk Management Plan

The plan for the risk management activities to be conducted on the SRS development process. The plan includes:

a) definition, or reference to, the risk management procedures to be applied, covering risk identification, risk analysis and risk management;

b) details of the SRS development risks identified, including assessment of level of risk, risk reduction measures and the effect of each on safety;

c) details of the risk analysis results, including identification of best case, most likely and worst case development timescales;

d) details of the ongoing risk management and control activities, including frequency of re-assessment of risk.

**B.9** Software Verification and Validation Plan

The plan for the V&V activities to be conducted on the SRS. The plan should include:

a) definition of the roles, responsibilities, independence criteria and reporting lines of those involved in the V&V;

b) identification of the V&V activities to be conducted throughout the SRS development, their scheduling and identification of those responsible for performing each activity;

c) description of the tools, techniques and methods to be used in the V&V;

d) details of the strategy for each V&V activity, for example test case selection;

e) details of the procedures for reporting and resolution of anomalies that are found during V&V;

f) the criteria for acceptance of each SRS item;

g) procedures for dealing with defects in the V&V process;

h) definition of the processes for controlling and reporting the results of the V&V activities;

i) criteria for selection of coverage metrics and the selection of targets for each metric.

**B.10** Software Configuration Management Plan

The plan for the configuration management and control of the SRS development. The plan includes:

a) definition of the purpose, scope and general applicability of the plan;

**B.10** (Cont)

b) definition of applicable standards (including this Standard and Def Stan 05-57) specifications and procedures applicable to the software configuration management;

c) identification of the organization and responsibilities relating to configuration management;

d) details of the definition and identification of configuration items, including naming conventions, version numbering and establishment of baselines;

e) details of configuration control, change requests, the process for implementing changes and the means of preserving the integrity of baselines and configuration items;

f) details of the tool(s), techniques and methods to be used for configuration management, including reference to procedures for the use of tools;

g) procedures for collecting, recording, processing and maintaining data relating to the configuration status;

h) requirements, procedures and schedules for the conduct of configuration management audits;

i) definition of software archive, retrieval and release processes, including release authorisation and requirements for data retention.

**B.11** Software Configuration Record

The record of the configuration of the completed SRS. The record includes, for each component of the SRS:

a) the name, version number and creation date;

b) details of the tools, their versions and configuration, used in the generation of the component;

c) identification of the documentation (including issue numbers) and assurance records relevant to the particular version of the component.

**B.12** Code of Design Practice.

The definition of the procedures, methods and techniques to be used in the development of the SRS. The Code of Design Practice includes details of:

a) the formal methods to be used during the specification process;

b) the methods to be used for preliminary validation of the Software Specification;

c) the formal methods to be used during the design process, and the way in which the Software Designs developed using these formal methods are to be formally related to the Software Specification;

**B.12** (Cont)

d) the policy on whether formal arguments will be presented as rigorous arguments or formal proofs and the cases where each should be applied;

e) the degree of detail required in rigorous arguments, and procedures by which proof obligations are to be discharged;

f) procedures for resolving any differences of interpretation of requirements and feeding back incomplete and inconsistent requirements to the originators of the Software Requirement;

g) procedures for ensuring that any requirements that evolve during the course of the development (derived requirements) are fed back to the Design Authority;

h) guidance on the production of the English commentaries on the Software Specification and the Software Design;

i) the structured design methods to be used in the specification and design processes and procedures for their use;

j) the implementation language and the way in which the source code is to be formally related to the Software Design;

k) the checks required to ensure that the requirements of the implementation language hold, including checks for conformance with any defined subset, and details of the means by which all such checks will be made;

l) the coding practices to be followed, including detailed requirements for coding as style, control structures and naming conventions;

m) the tools to be used during the SRS development process, including the compilation system and those tools used to support the use of formal methods and structured design methods;

n) procedures to be followed for using tools during the SRS development;

o) the means by which any known errors or inconsistencies in the tools to be used will be circumvented;

p) definition of, or reference to, the procedures to be followed for the conduct of reviews;

q) the failure assumptions which are to be protected against by defensive programming measures and the course of action to be taken if failures occur;

r) change control procedures, including the course of action to be taken if faults in the SRS are found during development and procedures defining the extent of rework necessary;

**B.12** (Cont)

s) the method, extent and scope of the means for achieving non-functional requirements;

t) the approach to be taken for fault tolerance, reliability, defensive code, accuracy, capacity, user interface, maintainability, software configuration and version identity.

**B.13** Software Specification

This document is produced by the Software Design Authority. It contains:

a) a formal specification of the functions, safety functions and safety properties of the SRS, written in one or more formal notations;

b) specification of non-functional properties of the SRS, written either in one or more formal notations, or in conventional (non-formal) notations;

c) English commentary on the formal specification.

**B.14** Specification Record

The Specification Record contains supporting documentation detailing the development and verification of the Software Specification. It contains:

a) minutes of any design meetings and design notes describing possible alternatives for the Software Specification;

b) minutes of all design reviews conducted on the Software Requirement and the Software Specification, including evidence of completion of actions;

c) all proof obligations constructed to verify the Software Specification and all formal arguments which discharge the proof obligations. Evidence of review and checking of the proof obligations and formal arguments;

d) the listing of the executable prototype(s);

e) all proof obligations constructed to demonstrate the mapping between the executable prototype(s) and the Software Specification and all formal arguments which discharge the proof obligations;

f) evidence of review and checking of the proof obligations and formal arguments;

g) the Executable Prototype Test Specification;

h) all test documentation for the executable prototype tests, including test data (inputs and expected outputs), test results, test descriptions, any harnesses or stubs required to execute the tests;

i) evidence of review of the executable prototype tests;

**B.14 (Cont)**

j) change history of the Software Specification and final configuration state.

**B.15** Software Design

The Software Design is produced by the Software Design Authority. It may consist of several levels of documentation, and may consist of several separately documented subsystems. The Software Design contains:

a) A policy for handling the non-functional aspects of the design, including the method, extent and scope. The policy addresses: fault-tolerance, reliability, defensive code, accuracy, capacity, user-interface, maintainability, software configuration and version identity.

b) A formal design of the functions, safety functions and safety properties of the SRS, written in one or more formal notations.

c) A design of the non-functional properties of the SRS, written either in one or more formal notations, or in conventional (non-formal) notations, including detail of the final implementation of the design on the target hardware.

d) English commentary on the formal design.

e) Change history of the Software Design and code and final configuration state.

**B.16** Design Record

The Design Record contains supporting documentation detailing the development of the Software Design and code, and the verification of the Software Design. It contains:

a) minutes of any design meetings and design notes describing possible alternatives for the Software Design and code;

b) minutes of all design reviews conducted on the Software Design and code, including evidence of completion of actions;

c) all proof obligations constructed to verify the formal design and all formal arguments which discharge the proof obligations. Evidence of review and checking of the proof obligations and formal arguments;

d) the static analysis of the code, including evidence of review of the static analysis and all anomalies found and actions taken as a result;

e) results of performance analysis;

f) the object code verification, including evidence of review of the object code verification and all anomalies found and actions taken as a result.

**B.17** Code

Full listings of the source and object code, including the run-time system and any command procedures necessary to build the code.

**B.18** Test Record

The Test Record encompasses the following items:

**B.18.1** Test Specification. The plan for each phase of testing (unit, integration, system and validation). For each phase, the Test Specification records the purpose and scope of all tests or sets of tests, and describes the environment required.

**B.18.2** Test Schedule. The detailed descriptions of the tests for each phase of testing (unit, integration, system and validation). For each phase, the Test Schedule records the inputs and expected outputs for all the tests. It also contains all stubs or harnesses required to run the tests.

**B.18.3** Test Report. The outputs and results of the tests for each phase of testing (unit, integration, system and validation). For each phase, the Test Report records the actual outputs obtained and the comparison of the expected and actual outputs. It details test failures and actions taken as a result. It also includes the coverage metrics achieved during testing and justification where these fall short of the targets. The coverage metrics include coverage of the requirements and traceability to the requirements.

**B.19** Acceptance Test Specification

The specification for the SRS acceptance tests. The specification includes:

a) identification of each test;

b) a high level description and explanation of the purpose of each test;

c) identification of the requirement within the Software Requirement that each test is intended to exercise;

d) description of the required test environment, including any stubs and harnesses and the means of checking results;

e) the test conditions and input data (including user options) to be used for each test;

f) the expected results and pass/fail criteria for each test.

**B.20** Acceptance Test Report

The report on the results of the SRS acceptance testing. The report includes:

a) a summary of the results of each test;

b) reference to the location of the detailed results;

**B.20** (Cont)

c) details of the comparison of the expected and actual results of each test;

d) a statement of whether each test has passed or failed;

e) analysis of all test failures and recommendations for any necessary action.

**B.21** User Manuals

Manuals providing instruction to users of the SRS, covering installers, operators and end users. The User Manuals include:

a) the version(s) of the software (and/or system) to which the manuals relate;

b) details of any restrictions and limitations in the use of the software/system;

c) description of the requirements and function of the software/system;

d) details of the safety functionality of the software/system, including any user overrides;

e) instruction on the installation and initialisation of the software;

f) instruction on user actions necessary for safe operation of the software/system;

g) the procedures for and format of any parameters or data to be entered by the user;

h) examples of the use of the software/system;

i) explanation of all error messages and advice on correction.

**B.22** Software Maintenance Plan

The plan for in-service support and software modification activities following delivery of an initial release of the SRS. The plan includes:

a) details of any user support requirements and processes;

b) details of the reporting procedures for in-service anomalies, including requirements for action for those anomalies with safety implications;

c) SRS change control methods, including configuration control issues associated with interfacing changed code and new assurance activities with existing code and assurance activities;

d) description, or reference to, the methods, tools and techniques to be used for software modification;

e) certification and acceptance procedures for SRS following modification.

SRS Certificate of Design

**PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE**

**CERTIFICATE OF DESIGN**

**For Safety Related Software**

| | |
|---|---|
| Certificate Number | *[unique identification number]* |
| Safety Related Software Identification | *[name]* |
| Safety Related Software Configuration Version/Mark/Model | *[version number]* |
| Design Authority | *[name]* |
| Parent System/Subsystem Identification | *[name]* |
| Parent System/Subsystem Configuration Version/Mark/Model/Build Standard | *[version number]* |
| Contract Number | *[contract reference]* |

We the designers of the above software hereby certify that :

1. The software complies with :
   (a) Defence Standard 00-55 issue *[issue number*] as qualified in the:
      i. Software Quality Plan [*document reference and issue number*].
      ii. Software Safety Plan [*document reference and issue number*].
   (b) Software Requirements Specification [*document reference and issue number*].

2. Evidence of compliance is provided in the Software Safety Case [*document reference and issue number*] and the Software Safety Records Log [*document reference and issue number*].

3. The software is suitable for use as defined in *[title and reference to appropriate user manuals]* which includes details of any restrictions or limitations for use.

Signatories:

Software Design Authority:

Signature:
Name:
Position:
Date:

Software Project Safety Engineer (if different):

Signature:
Name:
Position:
Date:

Software Developer (if different):

Signature:
Name:
Position:
Date:

As Independent Safety Auditor, I hereby certify that the work has been carried out in accordance with the Software Safety Plan and to my satisfaction:

Signature:
Name:
Position:
Date:

Tailoring Guide of This Standard Across Differing Safety Integrity Levels

## **D.1** Introduction

This annex defines the interpretation of the main text of the standard for software of safety integrity levels S1 to S3. The interpretation is presented as tailoring guide, in the form of a table, showing the application of each subclause for software of each safety integrity level. Much of the main text of the standard contains requirements that are considered to be 'good standard practice' for all SRS of all safety integrity levels and hence, for these requirements, the table shows no variation with integrity level. In other cases an activity may be required for all safety integrity levels, but the extent of the activity or the amount of evidence may be reduced with reducing integrity levels. The table is intended to provide developers with flexibility for the development of software of lower integrity levels and permits them to justify the development approach taken.

## **D.2** Application of this Standard

**Table D.1** provides details of the requirements for compliance with each of the clauses in Part 1 for the production of software of safety integrity levels S1, S2, S3 and S4 (although the Standard in its entirety applies to S4, this level has been included for completeness).

In the interests of brevity, where all parts of a clause or subclause have the same compliance requirements then they are not listed individually. The categories used are:

M Must be applied.

J1 Justification is to be provided if the clause or part of the clause is not followed. The justification may indicate that negligible improvement in integrity of the software will be gained from compliance, or that the cost of compliance is excessive for the benefit gained.

J2 Justification for the omission or non-compliance of the roles, methods, products or activities described within the clause is to be provided, although detailed justification of non-compliance with the individual clause or part of the clause is not required. The justification may indicate that compliance would not significantly improve the integrity of the software, or that the cost of compliance is excessive for the benefit gained.

Table D.1. Tailoring Guide of this Standard across Differing Safety Integrity Levels

| **Clause** | **S1** | **S2** | **S3** | **S4** | **Comments** |
|---|---|---|---|---|---|
| SECTION TWO. Safety Management | | | | | |
| 5 Safety Management Activities | N/A | N/A | N/A | N/A | |
| 6 Software Safety Plan | | | | | |
| 6.1 | M | M | M | M | Perhaps less detail for less critical or smaller systems |
| 6.2 | J2 | J1 | M | M | |
| 6.3 | J2 | J1 | M | M | |
| 6.4 | J2 | J1 | M | M | |
| 7 Software Safety Case | | | | | |
| 7.1 General | M | M | M | M | The extent of information provided will be less for lower SILs |
| 7.2 Development of the Software Safety Case | | | | | |
| 7.2.1 | J2 | J1 | M | M | |
| 7.2.2 | J2 | J1 | M | M | |
| 7.2.3 | J2 | J1 | M | M | |
| 7.2.4 | J2 | J1 | M | M | |
| 7.2.5 | J2 | J1 | M | M | |
| 7.2.6 | M | M | M | M | |
| 7.2.7 | M | M | M | M | |
| 7.2.8 | M | M | M | M | |
| 7.3 Safety arguments | | | | | Role of different types of evidence varies with SIL. Some discussion in Part 2 |
| 7.3.1 | J2 | J1 | M | M | |
| 7.3.2 | | | | | |
| 7.3.2(a) | J2 | J1 | M | M | |
| 7.3.2(b) | J1 | J1 | M | M | |
| 7.3.3 | J2 | J1 | M | M | |
| 7.3.4 | J2 | J1 | M | M | |
| 7.3.5 | J2 | J1 | M | M | |
| 7.4 Justification of SRS development process | | | | | The extent of information provided will be less for lower SILs |
| 7.4.1 | M | M | M | M | |
| 7.4.2 | M | M | M | M | |
| 7.4.3 | J1 | J1 | M | M | |
| 7.4.4 | J1 | J1 | M | M | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 7.4.5 | J1 | M | M | M | |
| 7.4.6 | J2 | J1 | M | M | |
| 7.4.7 | J1 | M | M | M | |
| 8 Safety Analysis | M | M | M | M | |
| 9 Software Safety Records Log | M | M | M | M | The amount of information for S1 and S2 will be less than for S3 and S4. If the Software Safety Case is straightforward the information in the Software Safety records Log could be incorporated in the Software Safety Case |
| 10 Software Safety Reviews | M | M | M | M | These could be merged with general project reviews for S1 and S2 |
| 11 Software Safety Audits | | | | | |
| 11.1 | M | M | M | M | |
| 11.2 | J1 | J1 | M | M | At S1 and S2 the Software Safety Audit Plan might be included in the Software Safety Plan |
| 11.3 | J1 | J1 | M | M | At S1 and S2, the audits might be recorded in the Software Safety Records Log |
| SECTION THREE Roles and Responsibilities | | | | | |
| 12 General | M | M | M | M | |
| 13 Design Authority | M | M | M | M | |
| 14 Software Design Authority | M | M | M | M | |
| 15 Software Project Manager | M | M | M | M | |
| 16 Design Team | M | M | M | M | |
| 17 V&V Team | | | | | |
| 17.1 | M | M | M | M | |
| 17.2 | J2 | J1 | M | M | Degree of independence may vary |
| 17.3 | M | M | M | M | |
| 17.4 | M | M | M | M | |
| 17.5 | M | M | M | M | |
| 18 Independent Safety Auditor | | | | | |
| 18.1 | M | M | M | M | |
| 18.2 | M | M | M | M | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 18.3 | J1 | J1 | M | M | Software Safety Audit Plan may be part of Software Safety Plan at S1 or S2 |
| 18.4 | J1 | J1 | M | M | Level of auditing may be reduced at S1 or S2 |
| 18.5 | J1 | J1 | M | M | Software Safety Audit Report may be part of Software Safety Records Log at S1 or S2 |
| 18.6 | M | M | M | M | |
| 19 Software Project Safety Engineer | M | M | M | M | |
| SECTION FOUR. Planning Process | | | | | |
| 20 Quality Assurance | | | | | |
| 20.1 | M | M | M | M | |
| 20.2 | M | M | M | M | |
| 20.3 | M | M | M | M | |
| 21 Documentation | | | | | |
| 21.1 | M | M | M | M | |
| 21.2 | M | M | M | M | |
| 21.3 | J1 | M | M | M | |
| 21.4 | M | M | M | M | |
| 21.5 | M | M | M | M | |
| 21.6 | M | M | M | M | |
| 21.7 | M | M | M | M | |
| 22 Development Planning | | | | | |
| 22.1 | M | M | M | M | |
| 22.2 | M | M | M | M | |
| 22.3 | M | M | M | M | |
| 22.4 | M | M | M | M | |
| 23 Project Risk | | | | | |
| 23.1 | M | M | M | M | |
| 23.2 | J1 | J1 | M | M | Software Risk Management Plan may be part of the Software Quality Plan for S1 or S2 |
| 23.3 | M | M | M | M | |
| 24 Verification and Validation Planning | | | | | Extent of Software V&V Plan may reduce with SIL as extent of V&V activities reduced with SIL |
| 24.1 | M | M | M | M | |
| 24.2 | M | M | M | M | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 24.3 | M | M | M | M | |
| 24.4 | M | M | M | M | |
| 25 Configuration Management | | | | | |
| 25.1 | M | M | M | M | |
| 25.2 | M | M | M | M | |
| 25.3 | M | M | M | M | |
| 25.4 | M | M | M | M | |
| 25.5 | M | M | M | M | |
| 25.6 | J1 | J1 | J1 | J1 | |
| 25.7 | M | M | M | M | |
| 25.8 | M | M | M | M | |
| 25.9 | M | M | M | M | |
| 25.10 | M | M | M | M | |
| 25.11 | M | M | M | M | |
| 25.12 | M | M | M | M | |
| 26 Selection of Methods | | | | | |
| 26.1 | M | M | M | M | . |
| 26.2 Required methods | | | | | |
| 26.2(a) | J2 | J1 | M | M | For lower SILs the extent of the functionality specified formally may be reduced. Detailed requirements defined under section 5 |
| 26.2(b) | J2 | J1 | J1 | M | The use of structured design methods is likely to be cost-effective at all SILs |
| 26.2(c) | J2 | J1 | J1 | M | All analysers should be used for S3 and S4, but requirement for semantic analysis may be relaxed for S1 and S2 |
| 26.2(d) | M | M | M | M | Dynamic testing is essential at all SILs, but the extent of the testing may vary. Detailed requirements defined under section 5. |
| 27 Code of Design Practice | | | | | The extent of the Code of Design Practice will reduce with SIL as the requirements for the methods reduce, but a Code of Design Practice is required at all SILs |
| 27.1 | M | M | M | M | |
| 27.2 | M | M | M | M | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 28 Selection of Language | | | | | |
| 28.1 High level language requirements | | | | | At lower SILs, language selection criteria other than integrity, eg tool support, performance, and portability may have greater influence than at higher SILs |
| 28.1(a) | J2 | J1 | M | M | |
| 28.1(b) | J2 | J1 | M | M | |
| 28.1(c) | J2 | J1 | J1 | M | |
| 28.1(d) | J2 | J1 | J1 | M | |
| 28.2 | M | M | M | M | |
| 28.3 | M | M | M | M | |
| 28.4 | M | M | M | M | |
| 28.5 Use of assembler | | | | | |
| 28.5.1 | J2 | J1 | M | M | More extensive use of assembler may be permitted at lower SILs |
| 28.5.2 | J2 | J1 | M | M | |
| 28.5.3 | M | M | M | M | |
| 28.5.4 | M | M | M | M | |
| 28.6 Compilation systems | | | | | |
| 28.6.1 | M | M | M | M | |
| 28.6.2 | | | | | |
| 28.6.2(a) | J1 | J1 | M | M | |
| 28.6.2(b) | M | M | M | M | The requirements for ISO 9001 and verification apply for all SILs, but the extent of the required verification is likely to reduce with reducing SILs |
| 28.6.3 | M | M | M | M | The extent of the required safety analysis is likely to reduce with reducing SIL. |
| 28.6.4 | M | M | M | M | |
| 29 Selection of Tools | | | | | |
| 29.1 | M | M | M | M | |
| 29.2 | M | M | M | M | |
| 29.3 | M | M | M | M | |
| 29.4 | M | M | M | M | |
| 29.5 Tool selection criteria | | | | | |
| 29.5.1 | M | M | M | M | |
| 29.5.2 | J1 | J1 | J1 | J1 | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 30 Use of Previously Developed Software | | | | | |
| 30.1 | M | M | M | M | |
| 30.2 Unreachable code | J1 | J1 | M | M | |
| 30.3 | J1 | J1 | J1 | J1 | |
| 30.4 Previously developed software not developed to the requirements of this Standard | M | M | M | M | The extent of the reverse engineering activity is likely to be less for software of lower SILs since the methods required are less rigorous |
| 30.5 In-service history | M | M | M | M | |
| 30.6 | M | M | M | M | |
| 30.7 | M | M | M | M | |
| 30.8 | M | M | M | M | |
| 30.9 | M | M | M | M | |
| 31 Use of Diverse Software | M | M | M | M | |
| SECTION FIVE. SRS Development Process | | | | | |
| 32 Development Principles | | | | | |
| 32.1 The lifecycle for SRS development | M | M | M | M | |
| 32.2 Production of the Software Specification, Software Design and code | | | | | |
| 32.2.1 | M | M | M | M | |
| 32.2.2 | | | | | |
| 32.2.2(a) | M | M | M | M | |
| 32.2.2(b) | M | M | M | M | |
| 32.2.2(c) | J1 | J1 | M | M | |
| 32.2.2(d) | J1 | J1 | M | M | |
| 32.2.2(e) | M | M | M | M | |
| 32.2.2(f) | M | M | M | M | |
| 32.2.2(g) | J1 | J1 | M | M | |
| 32.2.2(h) | J1 | J1 | M | M | |
| 32.2.2(i) | J1 | J1 | M | M | |
| 32.2.2(j) | J2 | J1 | M | M | |
| 32.2.3 | M | M | M | M | The extent of the content of the Specification Record and Design Record will vary with SIL. |
| 32.2.4 | M | M | M | M | The extent of the content of the Test Record will vary with SIL. |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 32.2.5 | M | M | M | M | |
| | | | | | |
| 32.2.6 | M | M | M | M | |
| 32.2.7 | M | M | M | M | |
| 32.3 Traceability | | | | | |
| 32.3.1 | M | M | M | M | |
| 32.3.2 | J1 | J1 | M | M | |
| 32.3.3 | M | M | M | M | |
| 32.3.4 | M | M | M | M | |
| 32.4 Verification of the Software Specification, Software Design and code | | | | | |
| 32.4.1 | J1 | J1 | M | M | |
| 32.4.2 | J1 | J1 | M | M | |
| 32.4.3 | J1 | J1 | M | M | |
| 32.4.4 | J1 | J1 | M | M | |
| 32.4.5 | J2 | J1 | M | M | |
| 32.4.6 | | | | | |
| 32.4.6(a) | J2 | J1 | M | M | |
| 32.4.6(b) | J2 | J1 | M | M | |
| 32.4.6(c) | J1 | J1 | M | M | |
| 32.5 Review | | | | | |
| 32.5.1 Review of formal verification | J2 | J1 | M | M | |
| 32.5.2 Review of Software Specification, Software Design and code | | | | | |
| 32.5.2(a) | M | M | M | M | |
| 32.5.2(b) | M | M | M | M | |
| 32.5.2(c) | M | M | M | M | |
| 32.5.2(d) | M | M | M | M | |
| 32.5.2(e) | J1 | M | M | M | |
| 32.5.2(f) | J2 | J1 | M | M | |
| 32.5.3 Review of anomaly correction | M | M | M | M | |
| 32.5.4 | M | M | M | M | |
| 32.5.5 | J1 | M | M | M | |
| 33 Software Requirement | | | | | |
| 33.1 General | | | | | |
| 33.1(a) | M | M | M | M | |
| 33.1(b) | M | M | M | M | |
| 33.2 | M | M | M | M | |
| 33.3 | M | M | M | M | |
| 34 Specification Process | | | | | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 34.1 | M | M | M | M | See clause 33 for variations with SIL |
| 34.2 Verification of the Software Specification | M | M | M | M | See clause 33 for variations with SIL |
| 34.3 Preliminary validation | | | | | |
| 34.3.1 | J1 | J1 | M | M | |
| 34.3.2 | J2 | J1 | J1 | M | |
| 34.3.3 | J2 | J1 | J1 | M | |
| 34.3.4 | J2 | J1 | M | M | |
| 34.3.5 | J1 | J1 | M | M | |
| 34.3.6 | J1 | J1 | M | M | |
| 34.3.7 | J1 | J1 | M | M | |
| 34.3.8 | J2 | J1 | M | M | |
| 34.4 Preliminary validation via formal arguments | | | | | |
| 34.4.1 | J2 | J1 | J1 | M | |
| 34.4.2 | J2 | J2 | J1 | M | |
| 34.5 Preliminary validation via executable prototype | | | | | |
| 34.5.1 | J2 | J1 | J1 | M | |
| 34.5.2 | J2 | J2 | J1 | M | |
| 34.5.3 | J2 | J2 | J1 | M | |
| 34.5.4 | J2 | J1 | J1 | M | |
| 34.5.5 | J2 | J1 | J1 | M | |
| 34.5.6 | J2 | J1 | J1 | M | |
| 34.5.7 | J2 | J1 | J1 | M | |
| 34.5.8 | J2 | J2 | J1 | M | |
| 34.5.9 | J2 | J1 | J1 | M | |
| 34.6 Checking and review of the specification process | | | | | |
| 34.6.1 | M | M | M | M | See clause 33 for variations with SIL |
| 34.6.2 | J1 | J1 | M | M | |
| 34.6.3 | J1 | J1 | M | M | |
| 35 Design Process | | | | | |
| 35.1 | M | M | M | M | See clause 33 for variations with SIL |
| 35.2 | M | M | M | M | |
| 35.3 | M | M | M | M | |
| 35.4 | J2 | J1 | M | M | |
| 35.5 Verification of the Software Design | M | M | M | M | See clause 33 for variations with SIL |
| 35.6 Checking and review of the design process | | | | | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 35.6.1 | M | M | M | M | |
| 35.6.2 | M | M | M | M | |
| 35.6.3 | M | M | M | M | |
| 35.6.4 | J1 | J1 | M | M | |
| 36 Coding Process | | | | | |
| 36.1 Coding standards | J1 | J1 | M | M | |
| 36.2 | M | M | M | M | |
| 36.3 | J2 | J1 | M | M | |
| 36.4 | J2 | J1 | M | M | |
| 36.5 Static analysis and formal verification | | | | | |
| 36.5.1 | J1 | J1 | M | M | |
| 36.5.2 | J2 | J1 | M | M | |
| 36.5.3 | J2 | J1 | M | M | |
| 36.5.4 | J1 | J1 | M | M | |
| 36.5.5 | J2 | J1 | M | M | |
| 36.5.6 | J2 | J1 | M | M | |
| 36.6 Object code verification | | | | | |
| 36.6.1 | J1 | J1 | M | M | |
| 36.6.2 | J2 | J1 | M | M | |
| 36.6.3 | J2 | J1 | M | M | |
| 36.6.4 | J2 | J1 | M | M | |
| 36.7 Checking and review of the coding process | | | | | |
| 36.7.1 | M | M | M | M | See clause 33 for variations with SIL |
| 36.7.2 | J1 | J1 | M | M | |
| 36.7.3 | J2 | J1 | M | M | |
| 36.7.4 | M | M | M | M | See clause 33 for variations with SIL |
| 36.7.5 | J2 | J1 | M | M | |
| 36.7.6 | M | M | M | M | |
| 36.7.7 | J2 | J1 | M | M | |
| 36.7.8 | J2 | J1 | M | M | |
| 36.7.9 | J2 | J1 | M | M | |
| 37 Testing and Integration | | | | | |
| 37.1 Principles of testing | | | | | |
| 37.1.1 | | | | | |
| 37.1.1(a) | M | M | M | M | |
| 37.1.1(b) | J1 | J1 | M | M | |
| 37.1.1(c) | M | M | M | M | |
| 37.1.1(d) | J1 | J1 | M | M | |
| 37.1.1(e) | J1 | J1 | M | M | |
| 37.1.2 | | | | | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 37.1.2(a) | M | M | M | M | |
| 37.1.2(b) | J1 | J1 | M | M | |
| 37.1.2(c) | J1 | J1 | M | M | |
| 37.1.2(d) | M | M | M | M | |
| 37.1.3 | M | M | M | M | |
| 37.1.4 | M | M | M | M | |
| 37.1.5 | J1 | J1 | M | M | |
| 37.1.6 | M | M | M | M | |
| 37.1.7 | M | M | M | M | |
| 37.1.8 | | | | | |
| 37.1.8(a) | M | M | M | M | |
| 37.1.8(b) | J2 | J1 | M | M | |
| 37.1.9 | J2 | J1 | M | M | |
| 37.1.10 | J1 | J1 | M | M | |
| 37.1.11 | M | M | M | M | |
| 37.1.12 | M | M | M | M | |
| 37.1.13 | M | M | M | M | |
| 37.2 | M | M | M | M | |
| 37.3 Unit, integration and system testing | | | | | |
| 37.3.1 | M | M | M | M | |
| 37.3.2 | J1 | J1 | M | M | |
| 37.3.3 | J1 | J1 | M | M | |
| 37.3.4 | J1 | J1 | M | M | |
| 37.3.5 | M | M | M | M | |
| 37.3.6 | M | M | M | M | |
| 37.4 Validation tests | | | | | |
| 37.4.1 | M | M | M | M | |
| 37.4.2 | M | M | M | M | |
| 37.4.3 | M | M | M | M | |
| 37.4.4 | J1 | J1 | M | M | |
| 37.4.5 | J1 | J1 | M | M | |
| 37.4.6 | J1 | J1 | M | M | |
| 37.4.7 | M | M | M | M | |
| 37.4.8 | J1 | J1 | M | M | |
| 37.4.9 | M | M | M | M | |
| 37.4.10 | M | M | M | M | |
| 37.4.11 | J1 | J1 | M | M | |
| 37.4.12 | M | M | M | M | |
| 37.4.13 | J1 | J1 | M | M | |
| 37.4.14 | J1 | J1 | M | M | |
| 37.4.15 | J1 | J1 | M | M | |
| 37.4.16 | M | M | M | M | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 37.4.17 | M | M | M | M | |
| SECTION SIX. Certification and In-service Use | | | | | |
| 38 Certification | | | | | |
| 38.1 | M | M | M | M | |
| 38.2 | M | M | M | M | |
| 38.3 | M | M | M | M | |
| 38.4 | M | M | M | M | |
| 39 Acceptance | | | | | |
| 39.1 | M | M | M | M | |
| 39.2 | M | M | M | M | |
| 39.3 | J1 | J1 | J1 | J1 | |
| 39.4 | M | M | M | M | |
| 40 Replication | | | | | |
| 40.1 | M | M | M | M | |
| 40.2 | M | M | M | M | |
| 40.3 | M | M | M | M | |
| 40.4 | M | M | M | M | |
| 41 User Instruction | | | | | |
| 41.1 | M | M | M | M | |
| 41.2 | M | M | M | M | |
| 42 In-Service | | | | | |
| 42.1 | M | M | M | M | |
| 42.2 | M | M | M | M | |
| 42.3 | M | M | M | M | |
| 42.4 | M | M | M | M | |
| 42.5 | M | M | M | M | |
| 42.6 | M | M | M | M | |
| 42.7 | | | | | |
| 42.7(a) | M | M | M | M | |
| 42.7(b) | M | M | M | M | |
| 42.7(c) | J1 | J1 | M | M | |
| 42.7(d) | M | M | M | M | |
| 42.7(e) | M | M | M | M | |
| 42.7(f) | M | M | M | M | |
| 42.7(g) | M | M | M | M | |
| 42.8 | J1 | J1 | J1 | J1 | |
| SECTION SEVEN Application of this Standard Across Differing Safety Integrity Levels. | | | | | |
| 43 Software of Differing Safety Integrity Levels | | | | | |
| 43.1 | M | M | M | M | |

Table D.1 (Cont)

| Clause | S1 | S2 | S3 | S4 | Comments |
|---|---|---|---|---|---|
| 43.2 | M | M | M | M | |
| 43.3 | M | M | M | M | |

Guidance on the Preparation of a Software Safety Case

## **E.1** Introduction

The objective of this document is to give guidance to readers who are preparing a Software Safety Case to the requirements of this Standard. The document covers:

(a) the development of a Software Safety Case;

(b) the structure and form of the safety arguments;

(c) the justification of the software development process;

(d) the contents of a Software Safety Case.

## **E.2** Development of the Software Safety Case

### **E.2.1** Objective

**E.2.1.1** A Software Safety Case is defined, in 7.1.2 of Part 1 of this Standard, as: *"a well-organized and reasoned justification, based on objective evidence, that the software does or will satisfy the safety aspects of the Software Requirement".*

**E.2.1.2** The main objective of a Software Safety Case should be to provide assurance that will convince a third party that the software will be adequately safe.

### **E.2.2** Software Safety Case lifecycle

**E.2.2.1** Part 1 of this Standard requires the Software Safety Case to be an up-to-date record of the current safety arguments and supporting evidence. Software Safety Case development should be an integral part of the software development process, and should evolve and be elaborated as the design progresses.

**E.2.2.2** The Software Safety Case should be formally issued to the MOD PM at agreed points in the project lifecycle. Projects following a conventional "waterfall" lifecycle should formally issue, as a minimum, the following versions of the Software Safety Case:

(a) the *preliminary* Software Safety Case which justifies the way in which the Software Safety Plan is expected to deliver Safety Related Software (SRS) that meets the safety requirements to the specified safety integrity level;

(b) the *interim* Software Safety Case which contains the evidence that the functional safety requirements are captured in the formal specification;

(c) the *operational* Software Safety Case which gives the complete set of safety arguments and evidence that the safety requirements of the delivered SRS have been met to the specified safety integrity level.

**E.2.2.3** The contents of the Software Safety Case should vary for these three versions. For example, since the preliminary Software Safety Case is written at the planning stage, it will only contain safety arguments about what is planned to be done rather than what has been done and achieved. The supporting evidence should be collected as the project is carried out and presented in later versions of the Software Safety Case. **E.5** gives details of the expected contents of each version of the Software Safety Case.

**E.2.2.4** Projects following different lifecycles from that assumed above should align the formally issued versions of the Software Safety Case with key project milestones. For example:

(a) Project A plans to deliver a Mark I system followed at a later date with a planned upgrade to a Mark II system. A minimum of six versions of the Software Safety Case should be issued (ie preliminary, interim and operational version for both deliveries).

(b) Project B has been planned to be carried out with a period of trials followed by in-service usage. A minimum of four versions of the Software Safety Case should be issued (ie preliminary and interim versions and an operational version before both trial and in-service usage).

**E.2.2.5** Once a system has entered service then the Software Safety Case should be kept up to date by the organization responsible for supporting the software.

New releases or upgrades to the software should result in the Software Safety Case being formally issued to the MOD for approval before being installed and used on any operational system.

**E.2.2.6** There are many systems currently in-service which do not have a System Safety Case nor a Software Safety Case. Upgrades of these systems for which this Standard is invoked should include a Software Safety Case for any new or changed software. The scope of the Software Safety Case should be agreed with the MOD PM as it may not be cost-effective to include existing SRS which does not change as a result of the upgrade.

**E.2.3** Formal issue of the Software Safety Case

**E.2.3.1** At various points in the project lifecycle the Software Safety Case should be formally issued to the MOD PM. Each formal release should involve the creation of a new baseline of the Software Safety Case. The baseline should include details of all relevant documents, analysis and test results etc. All changes made since the previous submission should be clearly marked.

**E.2.3.2** The Software Safety Case should be presented as a separate document (called the submission) which should refer out to the detailed supporting evidence. The submission should focus on presenting the main safety arguments which justify that the software does or will satisfy the safety aspects of the Software Requirement. The contents of the submission are described in **E.5**.

**E.2.3.3** The submission should be a stand-alone document which can be read by itself without constant reference to the supporting evidence. It is important, therefore, that the descriptive parts of the submission (eg system and design aspects, software description) should be presented at an appropriate level of detail to enable the safety arguments to be understood.

**E.2.3.4** The section on safety arguments should aim at capturing the fundamental relationships between the claims that are being made about safety and the supporting evidence which has been collected (some examples are given in **E.3**). This section should not include detailed evidence which detracts from the general flow of the document. Detailed evidence should be summarized and referenced instead.

**E.2.3.5** The Software Design Authority should ensure that the submission is consistent with the supporting evidence. The Independent Safety Auditor should check this aspect as part of the review of the Software Safety Case.

**E.2.4** Relationship to System Safety Case

**E.2.4.1** Safety is a system property, and it is impossible to present safety arguments about software in isolation. The Software Safety Case should, therefore, be part of and subordinate to the System Safety Case. Traceability from the Software Safety Case to the System Safety Case should be provided for safety requirements and safety arguments.

**E.2.4.2** Production of the Software Safety Case should follow the approach adopted for the System Safety Case. For example, one possible approach for a complex system might involve developing a top-level System Safety Case which is supported by lower level Subsystem Safety Cases. Each Subsystem which contains SRS should then have its own Software Safety Case. It may also be useful to prepare a top-level Software Safety Case which covers common aspects of all software in the system.

**E.2.5** Structure

**E.2.5.1** For extremely large or complex software systems, it may be necessary to structure the Software Safety Case in line with the architecture of the software. For example, a Software Safety Case for a large software system composed of large subsystems of differing safety integrity levels might be structured as:

(a) a set of Subsystem Software Safety Cases justifying the safety of each software subsystem;

(b) a top-level Software Safety Case justifying how the combination of subsystems is safe.

**E.2.6** Records management. The definition of a Software Safety Case given above states that the arguments about safety are to be based on objective evidence. The evidence (eg results of analysis, design and testing) will be collected as the project proceeds and the software is developed. Large amounts of evidence will often be collected and steps should be taken to organize this information in a way that makes access to it easy and efficient. Steps should also be taken to ensure the safe and secure storage of these safety records.

**E.2.7** Tool support. A minimum level of tool support needed to construct a Software Safety Case would be a modern word-processor coupled with a compatible diagram editor. The next level of tool support would to be use a modern documentation system which assists a reader in navigating easily between different documents. This sort of support would make the assessment and review of a Software Safety Case much easier. For example, a reader would be able to jump between the safety arguments in the Software Safety Case and the detailed evidence contained in other documents. Finally, there are some specialized safety case tools which are designed to support the systematic construction of safety cases and, in particular, safety arguments.

**E.3** Safety Arguments

**E.3.1** Introduction. A Software Safety Case consists of a number of different parts (eg a description of the system and design aspects, the software safety requirements and the current status). However, the heart of a Software Safety Case is composed of safety arguments which justify the claims that the software meets its safety requirements.

**E.3.2** Structure

**E.3.2.1** A safety argument can be thought of as relating some claim about meeting a safety requirement to the evidence which has been collected to support the claim. Each safety requirement should be covered in a safety argument. This Standard requires safety arguments of the achieved safety integrity levels of the SRS.

**E.3.2.2** A claim can be split into a number of sub-claims which, when all are shown to be true, ensure the truth of the top-level claim. It is important to justify that each combination of sub-claims is sufficient to support the higher-level claim. Each sub-claim can then be further split into more detailed sub-claims until the desired level of granularity is reached. A top-level claim might be decomposed into a hierarchy of sub-claims which could be viewed as "evidence" for the top-level claim - so the evidence used at one level of the argument can be:

(a) facts, for example based on established scientific principles and prior research;

(b) assumptions, which are necessary to make the argument (these may be assumptions about the real world or claims which will be substantiated later with supporting evidence);

(c) sub-claims.

**E.3.2.3** Much of the detailed evidence should be referenced out to supporting documents.

**E.3.2.4** Various ways of representing safety arguments have been developed. **Figure H.3** gives an example of part of a safety argument using a claim/sub-claim structure. The example has been deliberately kept simple to exemplify the concepts. Other approaches are more sophisticated and their richer notations may allow more convincing safety arguments to be made.

**E.3.2.5** The safety arguments could evolve over the lifetime of the project. Initially some of the sub-claims might actually be design targets, but as the system develops the sub-claims might be replaced by facts or more detailed sub-arguments based on the real system. Deviations in implementation should be analysed to see how this affects a sub-claim, and how changes in sub-claim affect the safety argument.

**E.3.2.6** If correctly designed, the higher levels of a safety argument should remain substantially the same as the design evolves. So the evolution of a safety argument at the top-level should be confined mainly to the changing status of the supporting sub-claims and assumptions. For example, there may be an assumption that a given tool will operate correctly, and this must be later supported by explicit field evidence or analysis. The status of the assumption would then change from "unsupported" to "verified" with a cross reference to the supporting document.

**E.3.3** Types of arguments

**E.3.3.1** The types of the safety argument used in a Software Safety Case will vary depending on the software design, the software development process chosen and the Software Safety Case strategy. A safety argument could be:

(a) *Analytical*, where the evidence is the results of some form of analysis, simulation or modelling and the argument is based on showing how the analytical results contribute to the claim. For example, the results of static analysis could support a claim that some software was free from certain classes of faults. This claim could then form part of an argument about the safe functional behaviour of that software. Another example would be a deterministic argument (or proof) constructed using the rules of predicate logic based on evidence in the form of axioms.

(b) *Empirical*, where the arguments rely on observation of the behaviour of the software. A particularly important form of empirical arguments are those based on testing. For example, the results of performance testing may be used to support part of an argument, supplemented by design analysis, that the real-time behaviour of the software is safe. Testing may also provide sufficient evidence to allow arguments based on statistical inference to be constructed that a system exceeds a particular reliability target with a certain degree of confidence.

(c) *Qualitative*, where the evidence might be adherence to standards, design rules, or guidance. In the case of design rules there could be some requirement for "defence in depth" where no single fault in the system could affect safety. The argument is then based on the achievement of some agreed acceptance criterion based on compliance to the stated rules.

In practice it is unlikely that any safety argument will be entirely deterministic or probabilistic. It may consist of a number of claims about specific properties of the system which may not necessarily be the same type.

**E.3.4** Robustness

**E.3.4.1** There is an accepted safety principle in developing safety cases in other industries (eg nuclear) that important safety arguments should be *robust*, ie a safety argument should be acceptable even if there are some uncertainties or possible errors in the arguments or supporting evidence. This is the main reason for requiring two independent safety arguments to support a top-level safety claim. In particular, Part 1 of this Standard requires two or more diverse safety arguments (ie different types of arguments based on independent evidence) of the achieved safety integrity level of the SRS including:

(a) analytical arguments that the SRS is logically sound;

(b) arguments based on observing (primarily through testing) the behaviour of the SRS.

**E.3.4.2** Part 1 of this Standard requires an analysis of the independence of the diverse safety arguments to identify any common factors in the arguments and to identify where the arguments are sensitive to variations in the underlying evidence. Guidance on carrying out this analysis is given in **annex F**.

**E.3.5** Strategies

**E.3.5.1** Development of safety arguments for a Software Safety Case should be done within the context of the safety arguments for the overall system. This section discusses how the required safety attributes of the system (eg reliability, availability and fail-safety) should be mapped onto safety attributes for the software.

**E.3.5.2** Options for the safety case argument

**E.3.5.2.1** In safety-related systems, the primary concern is with dangerous failures, and the safety argument should be focused on ways of inhibiting a dangerous failure. The various approaches to this can be characterized using the model for system failure shown in **figure E.1**.

**E.3.5.2.2** This follows the standard fault-error-failure model for software. A *fault* is an imperfection or deficiency in the operational software and is the primary source of software failures. However a program may operate as intended until some triggering input condition is encountered. Once triggered, some of the computed values will deviate from the design intent (an *error*). However the deviation may not be large enough (or persist long enough), so the system may recover naturally from the "glitch" in subsequent computations ("*self healing*"). Alternatively explicit design features (eg diversity, safety kernels) can be used to detect such deviations and either recover the correct value (*error recovery*) or override the value with a safe alternative (*fail-safety*).

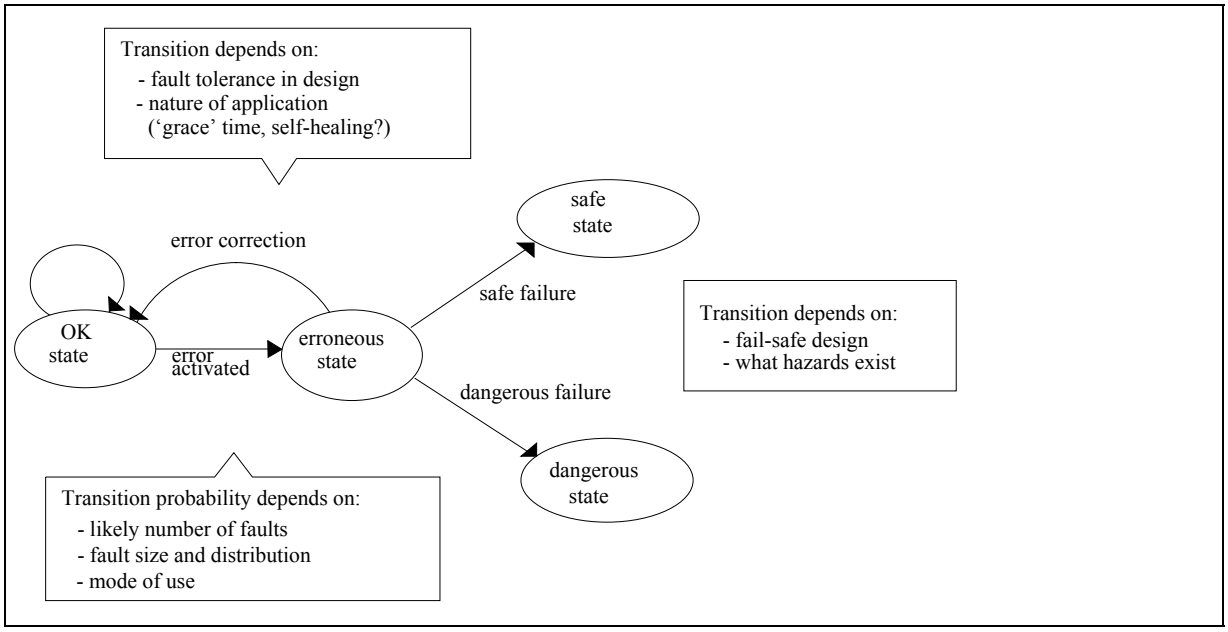



Figure E 1 : Model of system failure behaviour

**E.3.5.2.3** The overall approach to generating a safety argument involves:

(a) characterizing the safety arguments in terms of the transitions of the model;

(b) ensuring the implementation strategy is compatible with the safety argument;

(c) determining and evaluating the evidence to support the claims made about the transition probabilities in the model.

**E.3.5.2.4** This is a general model, and a particular safety argument can focus on claims about particular transition arcs. The main approaches are:

(a) A fault elimination argument can increase the chance of being in the "OK" state and can hence reduce or eliminate the *OK*→ *erroneous* transition. This is the reasoning behind the requirement to use formal methods in this Standard which essentially support a claim that the error activation rate was zero because there are no faults.

(b) A failure containment argument can strengthen the *erroneous*→ *OK* or *erroneous* → *safe* transition. An example would be a fail-safe design which quantifies the fail-safe bias. This, coupled with test evidence bounding the error activation rate, would be sufficient to bound the dangerous failure rate.

(c) A failure rate estimation argument can estimate the *OK*→ *dangerous* transition. The whole system is treated as a "black-box" and probabilistic arguments are made about the observed failure rate based on past experience or extensive reliability testing.

**E.3.5.2.5** It is also possible to apply the arguments selectively to particular components or fault classes, for example:

(a) A design incorporates a safety barrier which can limit dangerous failures occurring in the remainder of the system. The safety argument would then focus on the safety integrity of the barrier rather than the whole system.

(b) Different countermeasures might be utilized for different classes of fault. Each fault class then represents a separate "link" in the argument chain, and all fault classes would have to be covered to complete the argument chain. For example, design faults might be demonstrated to be absent by formal development, while random hardware failures are covered by hardware redundancy.

**E.3.5.2.6** The guidance above applies equally to continuously running systems and shutdown-systems. For example, it may be impossible to assure safety for some continuously running systems (eg an aircraft flight control system) by relying on a transition from an erroneous state to a safe state (ie aircraft stationary on the ground). For these types of systems, safety arguments should focus on error activation and error recovery. One of the main points of the model of failure behaviour given above is that it allows the differences in safety arguments between different types of systems to be seen clearly.

**E.3.5.3** Integrating the safety arguments with design and development. The previous discussion illustrates the key and closely coupled roles of the development processes and the design in formulating the safety arguments. Sometimes, the design and development approach

**E.3.5.3** (Cont)

is geared toward implementing the operational requirements; the need to *demonstrate* safety is only considered at a later stage. This can lead to considerable delays and additional assessment costs. The safety arguments should be an integral part of the design approach and the feasibility and cost of the safety arguments should be evaluated in the initial design phase. This "design for assessment" approach should help exclude unsuitable designs and enable more realistic design trade-offs to be made.

**E.3.5.4** Top-down derivation of safety requirements

**E.3.5.4.1** Safety is a property of the overall system rather than any given sub-component. The safety requirements should be identified by system-level safety analysis and may include functional behaviour, reliability, availability, fail-safety, maintainability, modifiability, security and usability. These safety properties or attributes should be mapped onto a set of functional and non-functional software safety requirements that have to be implemented to a given level of safety integrity. Software safety requirements should include both positive and negative safety requirements (eg "the rate of climb shall be kept within the following limits" and "this function shall be inoperative during landing").

**E.3.5.4.2** It should also be borne in mind that safety has to be maintained over the lifetime of the equipment. So the system design and the associated safety arguments should consider potential "attacks" on the design integrity over its lifetime (eg through normal operation, maintenance, upgrades and replacement). When these considerations are applied to individual subsystems (typically by applying hazard analysis methods), a set of *derived requirements* should be produced for the subsystems which are necessary to support the top-level safety goal. These derived requirements can be represented by *attributes* of the equipment which can affect plant safety. The most obvious attributes are reliability, availability and fail-safety, but there are other more indirect attributes including maintainability, modifiability, security and usability.

**E.3.5.5** Implementing the attributes

**E.3.5.5.1** When these attributes are implemented in a computer system design, it is difficult to consider the hardware and software elements in isolation. In order to implement the computer system attributes there may be new *derived requirements* which would typically add new functional requirements to the software and hardware components. For example, in order to maintain the overall integrity of the system, the software may rely on separate checking hardware (such as external watchdogs), while the hardware status might be monitored using software.

**E.3.5.5.2** This implies that the safety arguments for a computer system should start with the system architecture. This architectural safety argument should demonstrate why the design can implement the required behaviour. The architectural safety argument will claim that the safety requirements will be satisfied if the specified behaviour of the hardware and software components is correctly implemented.

**E.3.5.5.3** The safety argument at the software level therefore consists primarily of functional requirements (eg to implement required hardware diagnostics, or to service a watchdog) but some attributes may simply be targets to be achieved by the software design (eg worst case

**E.3.5.5.3** (Cont)

time response, or security levels). Since attributes can change (eg be transformed into functional requirements) as the design proceeds, it is important to maintain traceability of these attributes between the various levels of the overall Safety Case.

**E.3.5.5.4** The non-functional attributes at the system architecture level can lead to requirements for additional functionality within the software. These can be a major source of complexity for the software. In fact the software required for the actual control and protection functions could be quite modest compared with the code needed to implement these additional functions.

**E.3.5.5.5 Figure E.2** shows an example of how some system attributes might be implemented as software functions.

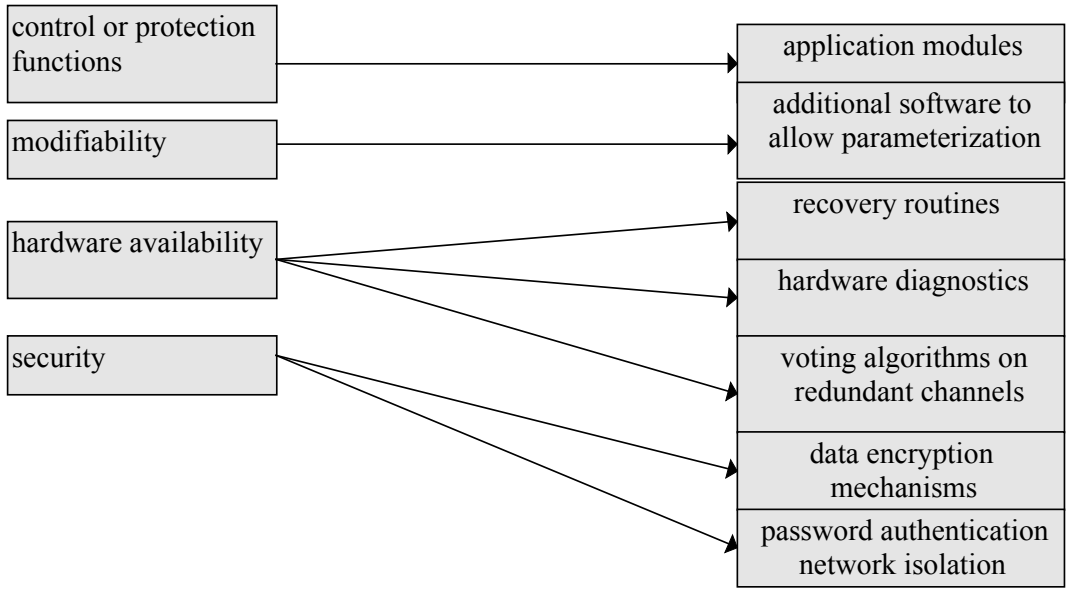



Figure E.1 : Relating Software Functions to System Attributes

**E.3.6** Example outline arguments

**E.3.6.1** The System Safety Case will also impose "derived requirements" on the software and these will include additional functional requirements (eg to support fail-safety and fault tolerance mechanisms). There may also be attributes such as timeliness which have been apportioned by the systems analysis, that have to be implemented and demonstrated at the software level. In addition, the system architecture design may impose additional *design constraints* (such as available memory) which must be respected before the system can function correctly. So the Software Safety Case should consist of a number of claims which link back to requirements and constraints imposed by the System Safety Case.

**E.3.6.2** **Table E.1** illustrates the types of claim that might be made for certain safety attributes. The text in italics refers to additional evidence which can be derived by Verification and Validation (V&V) activities when the system has been partially or wholly implemented. It is not claimed that this table presents a complete list or even a minimum list, since certain attributes (such as operability) may not be relevant for the given application, or may be addressed by some other element in the system architecture.

## E.4 Justification of the Software Development Process

### E.4.1 Introduction

**E.4.1.1** Software is not subject to failure modes caused by component degradation through physical processes and subsequent failure that affects hardware components. Instead, software failure modes are systematic in nature, that is they are due to mistakes made in the software development process. Furthermore, it is well understood that, in general, it is impractical to carry out a test programme which will exercise a software program completely. It is also impractical for many systems built to high safety integrity requirements to carry out operational testing to collect statistical evidence of safety integrity. Consequently, the nature of the software development process in terms of its capability to deliver software to the required safety integrity is an important aspect that must be considered and justified in the Software Safety Case.

Table E.1 : Example Outline Safety Arguments

| **Attribute** | **Claim** | **Argument** | **Evidence/Assumptions** |
|---|---|---|---|
| Functional correctness | There is no fault in the software implementation | Formal proof of specified safety properties<br>Formal proof that code implements its specification | The design is simple enough to be amenable to proof<br>Proof tool is correct (or unlikely to make a compensating error)<br>Compiler generates correct code (sub-argument might use formal proof, past experience, or compiler certification)<br>High quality V&V process<br>*Test results* |
| | Software reliability exceeds system requirement | Reliability can be assessed under simulated operational conditions | *Statistical test results* |
| Timeliness | The system will always respond within the specified time constraints | Software design ensures execution time is bounded and statically decidable | Maximum timing decided by static code analysis |
| | | Maximum time less than limit | *Dynamic tests of worst case time response* |
| Memory Constraints | The system will always have sufficient memory to continue operation | Software design is such that the memory is bounded and statically decidable<br>Maximum memory use is less than limit | Analysis of memory usage<br>*Stress testing of system* |
| Tolerance to hardware random failure | Identified hardware failures (computer system and its interfaces) are tolerated or result in a fail-safe response | Interface faults are software detectable (eg via redundancy or encoding), Internal failures are detectable and fail-safe | All failure modes have been identified<br>*Fault injection tests to check response* |
| Tolerance to overload | Demands in excess of the specified rates will result in a safe response | Design can detect overload conditions and either maintain a degraded service or perform a fail-safe action | There is sufficient processing power to cope with credible levels of overload<br>*Overload tests* |

Table E.1 (Cont)

| **Attribute** | **Claim** | **Argument** | **Evidence/Assumptions** |
|---|---|---|---|
| Maintainability | Parameter adjustments can be made without affecting safety | Software imposed limits ensure parameters remain in the safe range | Systems level analysis of allowable safe ranges<br>*Validation tests* |
| Operability | System is robust to faulty operator actions<br>System is designed to minimize user error | Design conforms to human factors standards<br>Actions checked for safety implications (eg software safety interlocks) | Interface prototyping<br>*Validation tests* |

**E.4.1.2** The Software Design Authority should justify how the software development process will deliver SRS to the required safety integrity. The justification should include:

(a) a safety analysis of the software development processes (including the methods, tools and human tasks involved);

(b) an argument for the inclusion of any previously developed software;

(c) an assessment of the performance of the software development in terms of measurements taken during the project;

(d) an analysis of historical data on the safety integrity of software developed using the proposed or similar processes.

**E.4.1.3** This clause gives detailed guidance on each of these parts of the justification in turn.

**E.4.2** Process safety analysis

**E.4.2.1** Introduction

**E.4.2.1.1** Process safety analysis investigates the ways in which a software development process could fail and as a result cause a potentially dangerous software fault. The analysis identifies the measures taken to prevent this happening and the defences employed to detect and remove the effects of such failures. Weaknesses, omissions and duplications in the software development process should be exposed by this analysis and these problems addressed through process improvement activities.

**E.4.2.1.2** The analysis should look at the ways in which the individual elements (ie the methods, tools, support software and tasks) of the overall software development project can fail and analyse how these failures could lead to potentially hazardous faults in the SRS. The analysis should consider ways in which these failures might:

(a) introduce faults (specifically dangerous faults) in the delivered software;

(b) allow faults to go undetected;

(c) threaten the independence of safety arguments;

(d) threaten the quality of the evidence used to support safety arguments;

(e) increase project risks and so threaten safety;

(f) violate the principle of reducing safety risks to As Low As Reasonably Practicable (ALARP).

**E.4.2.1.3** Process safety analysis can be used to support the justification for the Software Safety Case that the software development process is adequate as a means of developing software with the desired safety integrity. The analysis is normally only carried out at the start of the project and its results should be incorporated into the preliminary Software Safety Case. The analysis should be repeated if significant changes are subsequently made to the

**E.4.2.1.3** (Cont)

software development process during the project.

**E.4.2.1.4** The results of the analysis should be confirmed in the interim and operational Software Safety Cases which should contain evidence of conformity to the proposed process. Problem areas should be highlighted and corrective actions identified. The operational Software Safety Case should also include an assessment of the overall performance of the process, together with a justification of why any changes do not affect the safety goals for the system.

**E.4.2.1.5** The process safety analysis procedure described in this document is based on conventional hazard analysis techniques adapted to meet the needs of analysing the software development process.

**E.4.2.2** Concepts

**E.4.2.2.1** Before proceeding to describe the process safety analysis procedure, some key concepts and terminology are introduced.

**E.4.2.2.2** The overall software development process is viewed as consisting of a number of high-level activities (eg produce software specification, prepare module tests). These activities are typically broken down into more detailed activities (eg perform Z syntax check). At this level, the methods, tools and human tasks which make up the activities are considered in the analysis. Examples of these would be the Z specification language, the Z syntax and type checking tool, and the modelling of requirements in the Z language.

**E.4.2.2.3** A *process failure* is defined to be a failure of any method, tool or human task which results in the incorporation of a potentially hazardous fault in the SRS. Process failures also include the failures of testing or review activities to detect software faults. Examples of a process failure would be the failure of a development tool, a mistake made by a programmer in implementing a module specification or the use of an inadequate or inappropriate method.

**E.4.2.2.4** A *process hazard* is defined to be the end effect of some combination of process failures resulting in a potentially hazardous fault being incorporated into the SRS (for example, "Hazardous timing fault in the operational software"). In other words, a process hazard is a failure of the overall software development process to construct SRS with the required safety attributes.

**E.4.2.2.5** Process failures and hazards should be considered at an appropriate level to keep the analysis to a manageable size.

**E.4.2.3** Analysis procedure

**E.4.2.3.1** The main phases in carrying out a process safety analysis are defined in **table E.2**.

Table E.2 : Process Safety Analysis Phases

| Phase | Description |
|---|---|
| Process Definition | Model the software development processes in a suitable form for analysis. |
| Process Hazard Analysis | Analyse the software development process to identify the process failures and their effects. Identify the measures to prevent these failures happening and the defences needed to detect the results of any failures. |
| Review | Review the results of the analysis and implement any changes needed to the proposed software development process. |

**E.4.2.3.2** These phases are described further in **annex F**.

**E.4.3** Previously developed software

**E.4.3.1** Part 1 of this Standard requires evidence of the suitability of both previously developed software and Commercial Off-The-Shelf (COTS) Software. Evidence may take the form of:

(a) showing that the software has already been developed to 00-55 standards or some other appropriate standard (eg RTCA DO178B);

(b) third party safety certification of the software;

(c) carrying out a programme of reverse engineering to gather the necessary evidence of the safety integrity of the SRS;

(d) collecting evidence about the safety integrity of the software from its in-service usage.

**E.4.3.2** When considering the reuse of previously developed software or COTS software it should be borne in mind that software is prone to unexpected failure when used in a new environment or by a different set of users. For instance, an error in the way position co-ordinates are handled may cause the failure of previously safe equipment if it is moved to a different continent; or a progressive timing error may cause an anti-missile system to become inaccurate if it is operated by a service unit which carries out reinitialisation less frequently than previous users. Therefore, the similarity between the applications used to gather in-service safety integrity data and any proposed new application should be justified.

As a minimum, the following factors should be considered:

(a) data rates;

(b) functions used;

(c) operational mode;

**E.4.3.2** (Cont)

(d) size;

(e) complexity;

(f) throughput; and

(g) accuracy.

**E.4.4** Process data

**E.4.4.1** Measures of the performance of the software development process should be taken and compared against acceptance criteria based on historical norms.

**E.4.4.2** One of the most relevant types of measure of process performance in the context of developing SRS are those software metrics concerning faults. Measuring and analysing faults can play a useful part in running any type of software project. For example, an analysis of the reasons why faults are not detected close to the point of creation can be used to spot weaknesses in the software development process.

**E.4.4.3** Analysing the density of faults over a system should be used to identify those parts of the system where the density of faults is highest. These parts should be reviewed to determine if there are any underlying problems causing the high density of faults (eg the review should consider whether the design is too complex, if the requirements are badly understood or if more capable project staff should be used to rework these parts of the system).The analysis of fault density will often reinforce the project team's judgement about the areas of difficulty and should act as a catalyst for corrective action and process improvement.

**E.4.4.4** During the analysis of the data collected, any faults identified late in the development lifecycle which are (or are caused by) faults introduced in the early lifecycle should be highlighted since this indicates inadequacies in the review process, for example a fault picked up during testing which is a specification fault.

**E.4.4.5** Part 1 of this Standard requires acceptance criteria to be set for the measurements to be taken and these to be judged against historical norms. This implies that the software developer has access to such information from a set of such measurements taken over past projects. Software suppliers should set up databases to collect information on faults and populate it with data from past projects and retrospectively calculate historical norms.

**E.4.5** Historical data

**E.4.5.1** Part 1 of this Standard requires an analysis of historical data on the safety integrity of software developed using the proposed or similar software development processes. Historical evidence from previous projects and/or the published literature should be presented to show that highly dependable software has been successfully produced using such a development process.

**E.4.5.2** In the case of high integrity systems, failure rate data will be difficult to obtain unless there are a lot of systems in operation. Some of the guidance on error free operation (see **annex G**) and the reliance that can be put on this evidence should be taken into account.

**E.4.5.3** Evidence from past projects should be assessed by considering whether:

(a) the current system is similar to those produced on previous projects (eg similar size, complexity, performance requirements, nature and difficulty of the application);

(b) the software standards for SRS on the past projects are as demanding as 00-55;

(c) the safety integrity level of the previous applications is similar or greater than the current system;

(d) the staff used are at least as competent as the staff used on the previous projects.

**E.5** Software Safety Case Contents

**E.5.1** Introduction. This section gives guidance on the different parts that make up a Software Safety Case, the suggested contents of each part and how these parts should be developed as the project progresses.

**E.5.2** System and design aspects

**E.5.2.1** This part, in conjunction with the software safety requirements and software description, should give sufficient information to allow the safety arguments to be understood.

**E.5.2.2** The contents should be derived from the Software Requirement and include:

(a) an overview of the system architecture including the system boundaries and interfaces;

(b) an overview of the system functions;

(c) a brief description of the operating environment including both normal and abnormal modes of operation;

(d) a list of the main system safety requirements;

(e) a description of the system design approach (eg dual channel, fault tolerant hardware).

**E.5.2.3** This information should be available for the preliminary version of the Software Safety Case. It should be updated in later versions if any system or design aspect changes.

**E.5.3** Software safety requirements

**E.5.3.1** The contents of this part should be extracted from the Software Requirement and should describe the role that the software plays in ensuring safety. The contents should include a list of the functional and non- functional software safety requirements that have been derived from the equipment safety requirements by a process similar to that shown in **Figure E.2**. The safety integrity requirements for the software should also be documented

**E.5.3.1** (Cont)

together with any required software standards (eg configuration management, programming language definition). There should be traceability between the system and software safety requirements.

**E.5.3.2** This information should be available for the preliminary version of the Software Safety Case. The interim Software Safety Case should be updated if the process of writing a formal specification for the software results either in new safety requirements on the software being derived or the safety requirements being refined in some way. The operational Software Safety Case should incorporate any new or changed safety requirements.

**E.5.4** Software description

**E.5.4.1** The contents of this part should be derived from the Software Design and should describe the architecture of the software and how this contributes towards safety. The contents should include:

(a) an overview of the software architecture (eg main subsystems and their functionality);

(b) a description of the main design features of the software (eg real-time aspects, user interfaces, key algorithms, communication protocols, interfaces, database);

(c) the means by which software of different safety integrity levels is segregated (if relevant).

**E.5.4.2** This information may not be available and will not be complete for the preliminary and interim Software Safety Case. The status of the information made available in these two versions should be clearly indicated.

**E.5.5** Safety arguments

**E.5.5.1** This is the key part of the Software Safety Case. A safety argument should be given justifying how each software safety requirement has been or will be met. Part 1 of this Standard requires a justification of the achieved safety integrity level to be given by means of two or more independent safety arguments with, as a minimum, one argument based on analysis and one based on the results of testing. **E3** describes some approaches to building safety arguments.

**E.5.5.2** This section should also include safety arguments that all reasonable measures have been taken to reduce the software contribution to system hazards to an acceptable level.

**E.5.5.3** The contents of this section should be derived from the technical approaches described in the key planning documents (eg Software Safety Plan, Code of Design Practice and the Software V&V Plan).

**E.5.5.4** This part should focus on the high-level safety arguments and should summarize and reference the evidence upon which the safety arguments are based. A complete list of all assumptions used in constructing the safety arguments should be given. As safety arguments at the software level could form part of a safety argument at the system level, there should be traceability between the system and software safety arguments.

**E.5.5.5** The safety arguments should largely be in place for the preliminary Software Safety Case with later versions reflecting any refinements of the more detailed parts of the arguments that have occurred as the project proceeds. The detailed evidence which supports the safety arguments will not all be available for the preliminary nor the interim versions as many of the project activities which generate that evidence will not have been carried out. Each version of the Software Safety Case should clearly state the status of the supporting evidence (ie whether it has been or is still to be collected).

**E.5.6** SRS development process

**E.5.6.1** This part should give a justification that the software development process is adequate to achieve the required safety integrity level of the SRS. **E4** gives guidance on how this justification is to be given.

**E.5.6.2** This part should:

(a) briefly describe the main methods, tools and key project staff;

(b) summarize the results of the Process Safety Analysis;

(c) describe and justify the use of any previously developed software (including COTS software);

(d) provide a measure of the performance of the software development process (this should include the process data measures described in **E.4**;

(e) give the results of any safety or quality audits carried out;

(f) provide an analysis of historical data on the safety integrity of software developed using the proposed or similar software development processes.

**E.5.6.3** The results of safety and quality audits should be included with a note of any non-compliances and recommendations identified by the auditor and the status of any corrective actions taken or a justification for taking no action.

**E.5.6.4** The results of an internal company or independent assessment of software process maturity for developing SRS should be also be included. Clearly, an independent assessment would provide the stronger evidence.

**E.5.6.5** The results of the process safety analysis should be included in the preliminary Software Safety Case. The other information should be added as it becomes available.

**E.5.7** Current status This part should state progress against plan (Software Safety Plan and Software Development Plan) and discuss any outstanding issues which may affect safety such as:

(a) any safety requirements which are not being met

(b) any new or unresolved hazards which could potentially be caused by the software (eg a potentially hazardous software failure found during testing whose cause has not been found or which has not yet been fixed).

**E.5.8** Change history. This part should summarize any requirements changes which have affected the software and their impact on the safety. Any significant changes to the safety arguments since the previous issue of the Software Safety Case should be described.

**E.5.9** Compliance

**E.5.9.1** A statement of compliance against this Standard and any other required software standard should be given with a list of all non-compliances together with their status (eg agreed, under discussion).

**E.5.9.2** It should not be necessary to include a full clause-by-clause analysis of compliance although such an analysis should be done and recorded separately in the Software Safety Records Log.

**E.5.9.3** This part should be available in the preliminary Software Safety Case and kept up to date

**E.5.10** In-service feedback

**E.5.10.1** This part should summarize any relevant experience with the SRS or any part of it, from realistic operational testing, trials or in-service usage. This should include an analysis of any software failures on the safety of the equipment, addressing the possible consequences (ie hazards, accidents), the cause of the failure and any remedial action taken (this should include not only fixing any software faults but process improvement actions).

**E.5.10.2** This part will not normally form part of a Software Safety Case for a new system under development. However, if this is a development or enhancement of an existing system then this part should be completed and the relationship between the changes and the in-service feedback should be discussed (eg the reason for the software being changed might be to remove a new hazard which was overlooked in the existing system).

**E.5.11** Software identification. This part should identify the current release of the software citing a software release notice.

Process Safety Analysis Procedure

## F.1 Process Definition

**F.1.1** Introduction Process definition involves developing a process model of the overall software development process. A single model is needed so that the remaining steps in the analysis have a single source and can be carried out without having to refer to a number of different documents and plans. The process model will be a useful entity in its own right as it should enable the consistency and completeness of the overall software development process to be judged.

**F.1.2** Obtaining information on the process

**F.1.2.1** Information should be gathered on the software development process by analysing the project documentation particularly the plans, procedures and codes of practice that are produced as a result of project planning. It may be necessary to augment the information gathered from the documents with interviews to understand the intent and application of these documents. The information gathered should include details of the proposed methods, tools, procedures, standards, quality controls as well as details of the staff involved in each software development activity.

**F.1.2.2** As a minimum the following should be consulted:

(a) Software Safety Plan;

(b) Software Safety Audit Plan;

(c) Software Quality Plan;

(d) Software Development Plan;

(e) Software Risk Management Plan;

(f) Software Configuration Management Plan;

(g) Code of Design Practice;

(h) Software Verification and Validation Plan.

**F.1.3** Constructing the model

**F.1.3.1** Overall, the process model should identify:

(a) the inputs to each activity or task;

(b) the outputs created by each activity or task;

(c) how the inputs and outputs are being checked for faults;

(d) how it is being done, for example by whom, with what tools and techniques;

(e) when it is being done, for example at what stage in the project lifecycle.

**F.1.3.2** The scope of the model should include the software engineering, safety analysis, safety management and support activities (eg configuration management) whose failure could have an impact on the safety integrity of the SRS.

**F.1.3.3** The process model should be a summary of existing documentation, which should be cross-referenced. A clear mapping should be provided between the project documentation and the process model. For example, all the quality controls in the Software Quality Plan should be represented on the model.

**F.1.3.4** The example process model shown in **annex H** defines the overall software development process at the top level and then refines each top-level activity into a more detailed activity description.

**F.1.3.5** At the most detailed level, the process model should indicate how each input and output is checked for correctness and consistency, and what explicit checks are incorporated to assure safety - these are usually linked to specific safety-related attributes identified in the safety case (eg fail-safety, timeliness and reliability).

**F.1.3.6** It is recommended that the activities which support the main development and checking activities (eg configuration management, fault management) are modelled separately. As these activities interact with nearly all the main development and checking activities, it would be very confusing to include these activities within the main process model

**F.1.3.7** Finally, following construction of the model, it should be reviewed for completeness, internal consistency (eg matching of flows and common terminology) and any other checks as required by the method or notation used to construct the process model.

### **F.1.4** Process modelling techniques

**F.1.4.1** A variety of modelling techniques could be used to define the processes and their interdependencies. Some appropriate methods and notations are:

(a) business process re-engineering modelling approaches (eg Role Activity Diagrams);

(b) software design methods (eg data flow diagrams);

(c) project planning support (eg PERT and Gantt charts).

**F.1.4.2** It is likely that more than one technique will be needed to meet the aims and objectives of process safety analysis.

## **F.2** Process Hazard Analysis

### **F.2.1** Introduction

**F.2.1.1** The main steps in this phase are:

(a) process hazard identification;

**F.2.1.1** (Cont)

(b) process failure analysis.

**F.2.1.2** Each of these stages is described below.

**F.2.1.3** **Annex H** gives examples of parts of an analysis undertaken on the software development process of a project which is developing safety-critical software to the requirements of 00-55/1 (April 1991).

**F.2.2** Process hazard identification

**F.2.2.1** Process hazard identification requires the identification of the safety attributes relevant to the operational software and rewording each attribute into a hazard definition (an unwanted state of the software).

**F.2.2.2** **Table E.1** (see page **E-11**) gives a list of potential safety attributes that should be used as a basis for process hazard identification. This checklist should be supplemented by consideration of the safety requirements for the software and a review activity to ensure that all safety attributes (process hazards) have been identified.

**F.2.3** Process failure analysis

**F.2.3.1** Objective. At every stage in the software development process, faults can be introduced into the software products as a result of failures in the software development activities. Process failure analysis seeks to identify:

(a) how software processes can fail and the effects of the failures in terms of introducing or failing to detect potentially hazardous faults in the software products (ie process hazards);

(b) the measures that are taken to avoid these failures;

(c) the defences against these failures in terms of detecting and removing faults in the software products;

(d) the common causes of failures in the software development process.

**F.2.3.2** Process failure mode analysis

**F.2.3.2.1** The first step of process failure analysis should identify:

(a) the process failures which could occur and result in a process hazard;

(b) what measures have been taken to avoid process failures;

(c) what defences exist to detect the results of process failures so that remedial action may be taken to remove the faults found.

**F.2.3.2.2** Further guidance for the sorts of evidence that should be collected about the measures and defences is given in **annex J**.

**F.2.3.2.3** In carrying out this analysis, it is important to understand the main failure modes of the process and how these are caused. The main types of process failures are caused by:

(a) mistakes made by project staff in carrying out the creative, knowledge based and highly skilled tasks that form part of the software development process;

(b) the use of inadequate or inappropriate methods of carrying out these processes;

(c) the failures of tools which result in faults being introduced into a software product (eg a compiler fault) or failure of a checking tool to detect a fault in a product.

**F.2.3.2.4** Process failures which could result in or contribute to software faults should be identified by:

(a) analysing historical data of the type of faults that are found in subsequent checking activities and in operation;

(b) assessing project risks to give an indication that certain activities are vulnerable (eg because of lack of resources, insufficiently experienced personnel, novel tools).

Knowledge of process failures which occurred in past projects should be used as a starting point for identifying possible process failure modes as part of this step.

**F.2.3.2.5** The defences against process failures should be assessed by considering the impact of process failures on the software products that are produced in the development process. To do this, the different attributes that are required of a product and how these might be compromised should be examined. For example, in the specification, the process failures that affect the "correctness" attribute could be examined. These could include faults relating to syntax, internal consistency, completeness and validity.

**F.2.3.2.6** For each attribute, the initial defences within that phase and in "downstream" activities should be identified. The primary checks are applied to the product itself, while the "downstream" checks are applied to derived products (such as the complete program). **Table F.1** provides examples.

Table F.1: Example Defences

| **Attribute** | **Primary defences** | **Downstream defences** |
|---|---|---|
| Syntax | • Z syntax checker | |
| Internal consistency | • specification review<br>• Z proof obligations | • Design to specification refinement proof |
| Completeness | • tracing to the requirements<br>• animation of the specification | |
| Validity | • prior prototype implementation<br>• animation of the specification | • Validation tests |

**F.2.3.2.7** The analysis should be supported by subsidiary documents justifying the choice of specific methods and tools (eg codes of design practice and tool integrity assessments). Such subsidiary justifications can be reused for subsequent projects. The ALARP principle should be considered in this justification: it may be reasonable to introduce further defences if their cost is low, but it may be possible to argue that certain other techniques are disproportionately expensive for the improvement they bring.

**F.2.3.2.8** As with traditional, product-based hazard analysis, some means of classification of the impact of process failures should be used to focus the analysis. One way of classifying the effect of process failures illustrated in **table F.2**.

Table F.2. Classification of Impact of Process Failures

| **Category** | **Description** |
|---|---|
| Major | These seriously undermine the safety arguments and cannot be readily compensated for |
| Minor | These reduce assurance in the evidence but can be corrected or compensated for elsewhere |
| Negligible | The rest |

**F.2.3.2.9** A classification such as this should be used to prune the analysis at this relatively early stage so that low impact process failures are not considered any further thus leaving the analysis to concentrate on the more important process failures.

**F.2.3.2.10** The results of the analysis should then be summarized for example in a coverage matrix which shows what checking activities are applied to each development activity. The summary combined with the detailed analysis provide two contrasting levels of analysis. The summary is drawn up as a means of aiding the review of the overall process.

**F.2.3.3** Cause and effect analysis

**F.2.3.3.1** The next step is to draw up the detailed cause and effect relationships between the process failures and the process hazards identified in the earlier steps. Techniques such as fault tree analysis, event tree analysis and cause consequence diagrams may be appropriate.

**F.2.3.3.2** This step should result in a better understanding of the causes and defences relevant to each process hazard. Additionally it enables a comparison to be made between the level of defences provided for the various process hazards. For example, if fault tree analysis were used then it would be possible to calculate the cut sets for each process hazard. The nature of the elements making up each cut sets gives an indication of the strength in depth of the software development process and this could be one aspect to be considered in the review of the process hazard analysis.

**F.2.3.4** Common mode failure analysis

**F.2.3.4.1** The analysis should be supplemented by consideration of the potential sources of common mode process failure within the various development and checking activities. Common sources of failure could have a significant impact on the independence of top-level safety arguments.

**F.2.3.4.2** Within a Software Safety Case there is a requirement to provide diverse (ie independent) safety arguments of the achieved level of safety integrity of the SRS. Specifically, there are requirements to provide at least two diverse arguments derived from the results of analysis and testing.

**F.2.3.4.3** The process should be analysed to determine whether there is any commonality in the analysis (ie the main development activities) and testing (ie the main V&V activities) between:

(a) the inputs and outputs used (eg using the developer's specification for reliability testing);

(b) the methods and tools;

(c) the personnel (eg a developer involved in designing V&V tests).

**F.2.3.4.4** Where a common factor exists, a justification should be given as to why this is acceptable. For example there could be a common factor between the V&V and development activities as both are based ultimately on a common statement of requirements. However, it may be infeasible to provide two independent but equivalent statements of requirements and thus emphasis should be placed, in this instance, on carrying out measures to ensure the integrity of the statement of the requirements.

**F.3** Review

**F.3.1** The results of the process definition and process hazard analysis phases should be reviewed to assess the adequacy of the software development process as suitable for developing software of the required level of safety integrity.

**F.3.2** The review should take the form of an independent peer review involving key members of the project team together with the Independent Safety Auditor and the MOD Project Manager (or his/her specialist advisors). The review should take into account factors such as:

(a) compliance with best software engineering practice;

(b) compliance with the requirements of this Standard, Def Stan 00-56 and any other related safety standards;

(c) the number and strengths of the measures and defences employed; and

(d) the completeness, accuracy and objectivity of the supporting evidence.

**F.3.3** Compliance with best software engineering practice should take into account new technical advances which offer the ability, for instance, to remove certain classes of faults. For instance, it has recently become possible to detect via the use of an analysis tool all run-time exceptions for programs written in some languages.

**F.3.4** The review should consider whether adequate sets of measures and defences exist for all process hazards. For example, it is recommended that for each process hazard there are at least two defences: for example one defence based on a review activity and one based on an automated analysis or testing task.

**F.3.5** The review should seek to identify any inadequacies, non-compliances, omissions or duplications in the proposed software development process. Opportunities to improve the process should also be considered. The review should decide on the actions which need to be taken to resolve any problems. Records of the review should be taken and stored in the Software Safety Records Log and the results of the review summarised in the Software Safety Case.

Product Evidence

## G.1 Introduction

**G.1.1** Products are the deliverables from the software development process. This means items such as the source code, user documentation, specification and designs as well as all the outputs from the V&V activities.

**G.1.2** One of the most common ways of gathering evidence about a product is through testing. The programme of V&V activities carried out on a safety-related project will provide direct evidence of attributes such as reliability and performance. The unit, integration and system testing and most of the system validation testing activities called for by Part 1 of this Standard are widely practised and understood by software development organizations involved in the development of SRS and no further guidance is given in this document for collecting the evidence produced by these activities.

**G.1.3** Part 1 of this Standard does require, if feasible, a test programme to be run to collect statistically valid evidence of the safety and reliability properties of SRS. Consequently, a detailed discussion of the complex issues involved in setting up such a programme is given in the remainder of this section. Three different approaches (statistical/operational testing, reliability growth modelling and error free operation) to collect evidence of software reliability are given together with a discussion of the problems and limitations of these approaches.

**G.1.4** It is acknowledged that safety and reliability are different attributes. In particular, a safe system may not be a reliable one and vice-versa. However, many safety-related systems are required to achieve a certain level of reliability in order for them to be considered safe. The guidance on reliability is aimed at these types of systems.

## G.2 Statistical/Operational Testing

**G.2.1** Many claims that software is "sufficiently" reliable are based upon indirect evidence, for example of the quality of the development process. The only means of measuring the reliability of software directly depends upon being able to observe its operational behaviour for a sufficient time. Informally, if a program is observed to execute failure-free for long enough (or if a sufficiently small number of failures are observed during a long period of operation), it would be reasonable to accept claims that it is sufficiently reliable for its intended purpose. More precisely, such data allows precise quantitative assessments of the reliability to be made.

**G.2.2** With safety-related systems, it is rarely the case that real operation can be used to obtain this kind of data: there is a need to know that a system is sufficiently reliable before people are exposed to the risk of its operation. It will therefore be necessary to simulate operational use in a testing environment: such testing is called operational testing or statistical testing. The key idea here is that the selection of test cases is carried out in such a way that the probabilities of selection are the same as those in real operation, so that any estimates of reliability in the testing environment can be assumed also to estimate the operational reliability.

**G.2.3** Clearly, the difficulty of doing this will vary considerable from one application to another. In certain industries there is already considerable experience of creating realistic simulations of physical system behaviour - "iron birds" in the aircraft industry, for example - and it might be expected that in these cases it would also be possible to conduct accurate operational tests of software. It seems likely that many, if not most, SRS applications concern the control of well-understood physical systems, for which accurate simulation of behaviour is possible. This contrasts with many business systems, for example, where the nature of the use to which the product will be put can only be predicted imperfectly.

**G.2.4** Sequences of statistically representative test cases can be constructed in many different ways. In some applications, there may be similar products in operation, and scripts of operational use can be recorded directly. In others, it will be necessary to construct the operational profile - the probability distribution over the set of all inputs - indirectly. One way to do this is to partition the input space in some convenient way, so that the probabilities of the different partitions are known, and the probabilities of the inputs within each partition can be approximated. A recent example of this was the statistical testing of the software for the Primary Protection System (PPS) of the Sizewell B nuclear power station. Here eleven main scenarios were identified for which the PPS would be called upon to trip the reactor and keep it in a safe state following trip. An example of such a scenario is a major pipe break resulting in loss of coolant. Experts were able to assign probabilities to the different scenarios according to their knowledge of the wider plant, and historical data. The statistical testing was then the selection of many particular demands within these scenarios. Here the demands were chosen by randomly varying the parameters of each scenario: thus in the case of a pipe break, the size and location of the break would be chosen randomly, again using expert judgement for the probabilities of the different alternatives.

**G.2.5** This kind of operational testing can be used in two main ways - for reliability growth estimation, or for estimating reliability from evidence of failure-free working - as shown in the next sections. In both cases, it should be emphasized that the accuracy of the results will depend crucially upon the accuracy with which the simulation represents the statistical properties of the real operational environment.

## **G.3** Reliability Growth Modelling

During operational testing it is important to record the data accurately. This will be of two types: time, and counts of failure events. The time variable to be used will depend upon the application, and must be chosen with care. In process control applications, for example, the time will probably be actual plant operating time, and our reliability might be expressed as a rate of occurrence of failures, or a mean time to failure. In a safety system which only has to operate when (infrequently) called upon, such as the PPS above, time will be a discrete count of demands, and the reliability might be expressed as a probability of failure upon demand.

**G.3.1** Reliability growth models assume that, if failures occur during operational testing, the faults that cause these are fixed (or, at least, fixes are attempted). The models use the resulting succession of times between failures to estimate the reliability of the software and predict how this will change in the future on the assumption that fault-fixing proceeds in the same way.

**G.3.2** This is one of the most highly developed areas in software measurement, with a large scientific literature. Many different ways of representing the growth in reliability have been proposed over the years, but no definitive one has emerged. However, in recent years there have become available some powerful techniques that allow the accuracy of the reliability measures emanating from the testing of a particular program to be judged. Thus a user is able to measure reliability and be confident that the measures are accurate.

**G.3.3** For software required to meet high levels of safety integrity, however, reliability growth techniques have two serious limitations. In the first place, they are only feasible for measuring reliability when the levels needed are relatively modest, since there is a strong law of diminishing returns operating: for example, if a mean time to failure of x hours is the reliability target, then many times x hours on test will be needed before confidence can be gained that the target has been reached. Therefore this approach may only be practical for software with requirements for lower levels of safety integrity.

**G.3.4** Secondly, a conservative assumption should be taken that when a fix is carried out, a new program is created which should be evaluated afresh without using the data collected from its ancestors.

**G.4** Error Free Operation

**G.4.1** It is common, when assessing the reliability of SRS, to impose an additional condition that no known faults be present. Thus if a failure occurs on test, the fault will need to be identified and removed, and the software placed back on test until it has executed for a sufficiently long time without failure for it to be deemed sufficiently reliable.

**G.4.2** Relatively simple statistical techniques can be used to calculate the reliability that can be claimed for a particular period of failure-free working, under fairly plausible assumptions. Thus, for example, in the case of a safety system with a probability of failure on demand required to be better than $10^{-3}$ (this is approximately that needed for a reactor primary protection system), if the system survives about 5,000 tests then one could be more than 99% confident that it was good enough. All that is assumed here is that the successive demands during test are statistically independent of one another (and, of course, they are selected with probabilities that are the same as they would be in operational use).

**G.4.3** Once again, it needs to be emphasized that the levels of reliability that can be quantified with confidence in this way are relatively modest. Thus for a demand-based system, evidence of several times $10^{x}$ failure-free demands will be needed before a claim that the probability of failure upon demand is better than $10^{-x}$ can be justified with a high degree of confidence. There are similar limits to what can feasibly be claimed for continuously operating systems, such as control systems: for example it has been shown that if a system has survived t hours of operation without failure, there is only a 50:50

**G.4.3** (Cont)

chance of it surviving a further t hours. These limits are based, of course, upon certain assumptions about the failure processes - such as independence of successive demands, and "purely random" failures in the continuous case, but these are reasonably plausible. Even if these assumptions are not exactly correct, it seems clear that the arguments used give results that are "order of magnitude" right.

Process Safety Analysis Examples

## H.1 Introduction

**H.1.1** The examples used in this appendix are based on an MOD project called SHOLIS (Ship Helicopter Operating Limit Instrumentation System) which is developing safety-critical software to the full requirements of 00-55/1 (April 1991).

**H.1.2** SHOLIS is a real-time system designed to enhance the safety of shipborne helicopter operations. SHOLIS integrates ship heading and motion data with information concerning the helicopter and ship characteristics to determine if current operating conditions are within safe limits. If operating conditions are unsafe then audio-visual alarms are raised.

## H.2 Process Safety Analysis and the SHOLIS Software Safety Case

**H.2.1** This section shows how process safety analysis is used to generate the evidence needed to support claims made in the Software Safety Case about the adequacy of the software development process.

**H.2.2** The safety argument in **figure H.1** consists of a single top-level claim (**sil_claim**) which is refined into a hierarchy of sub-claims. The lowest level sub-claims, herein called 'base claims', are then justified by reference to evidence or assumptions. The evidence that can be used to support base claims is detailed in **annexes G** and **J.**

**H 2.3** The top-level claim is that the SHOLIS safety-critical software achieves Safety Integrity Level 4; this is shown to be dependent upon the validity of three sub-claims:

(a) safe timing behaviour is exhibited (**timing**);

(b) safe functional behaviour is exhibited (**func**);

(c) there is always enough memory available for continual operation (**mem**).

**H.2.4** These sub-claims are joined by an AND-gate indicating that the top-level claim depends on all three sub-claims being valid.

**H.2.5** For illustrative purposes figure H.1 only develops the claim for safe functional behaviour. The memory and timing claims are not developed but it should be noted that a real safety case must refine these claims fully.

**H.2.6** The claim of safe functional behaviour is shown by **figure H.1** to be dependent on the validity of either of the following sub-claims:

(a) correctness by construction ensures safe functional behaviour (**func.safe_construction**);

(b) testing demonstrates safe functional behaviour (**func.safe_testing**).

**H.2.7** These sub-claims are joined together by an OR-gate which indicates that the sub-claims are held to be independent and each is sufficient in its own right to justify the higher level claim. In order to justify this position, it is necessary to show that the two sub-claims are independent of one another. Common mode failure analysis which will be described later in this annex is one approach to investigating the independence of different claims. Note that it is at this level of the claim structure where the requirement in Part 1 of this Standard for two or more diverse safety arguments is satisfied by the SHOLIS Software Safety Case.

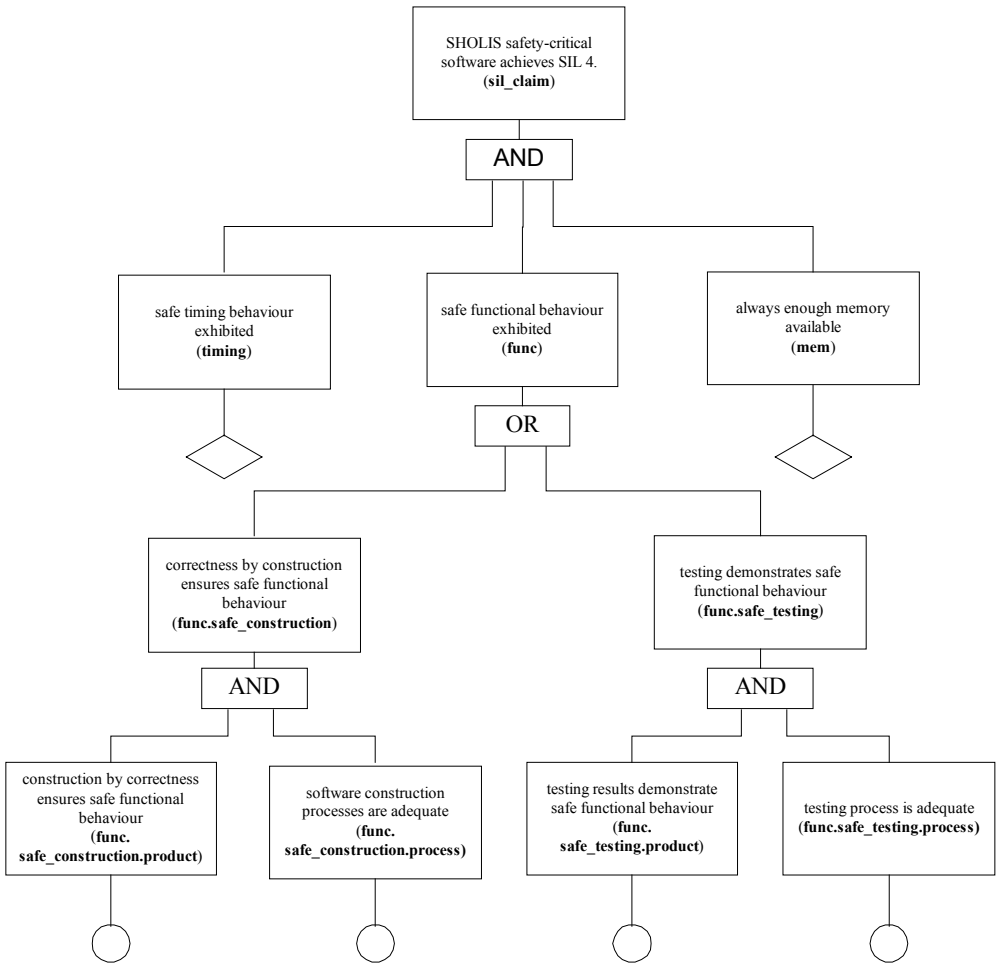



Figure H.1 : Claim Structure of part the SHOLIS Software Safety Case

**H.2.8** **Figure H.1** shows both the **func.safe_construction** and **func.safe_testing** claims as being dependent on two more detailed sub-claims: one concerned with evidence of the safety of the product gathered from carrying out software construction or testing activities; the other concerned with the adequacy of the process that gives rise to the evidence. The need for a combination of both product and process claims is best illustrated by example. The claim **func.safe_construction** is based, mainly, on the use of formal methods and static analysis to ensure correctness by construction. However, if, for example, a tool used to assist the static analysis was flawed then the analysis results may be valueless, consequently, a second sub-claim is required that the process by which the analysis was performed was itself adequate, ie the sub-claim **func.safe_construction.process**.

**H.2.9** The claims at the bottom of figure H.1 can be described as 'base claims' insofar as the SHOLIS Software Safety Case does not refine them further in order to keep the length of the example short. It is important to stress that in practice further refinements of the claim structure should be carried out to provide a more convincing safety argument.

**H.2.10** Once the base claims have been identified, the next objective is to provide evidence to substantiate them. If each base claim can be substantiated then an argument for the validity of the overall top level claim can be made.

## H.3 SHOLIS: Process Definition

### H.3.1 Introduction

**H.3.1.1** The first stage of process safety analysis is the production of a process definition via some form of process model. One possible approach to process definition will now be introduced using the SHOLIS project for illustration. Note that the full process definition for SHOLIS is not given here as it is too large to use as an example. Rather, a complete vertical slice of the project has been taken by examining a single SHOLIS development activity. The full SHOLIS process definition would be produced by repeating the steps for all project activities.

**H.3.1.2** The process model should capture the important characteristics of the development process while not becoming subsumed in huge amounts of detail. The process model should:

(a) capture inputs and output;

(b) capture resources (eg tools, methods);

(c) capture the role of human beings;

(d) reveal the process dynamics, for example if a process contains a high degree of iteration or parallelism then the model should show this.

**H.3.1.3** The example given (see **figure H.2**) here uses a combination of Role Activity Diagrams (RADs), Data Flow Diagrams (DFDs) and support tables (taken from the IDEF0 method)

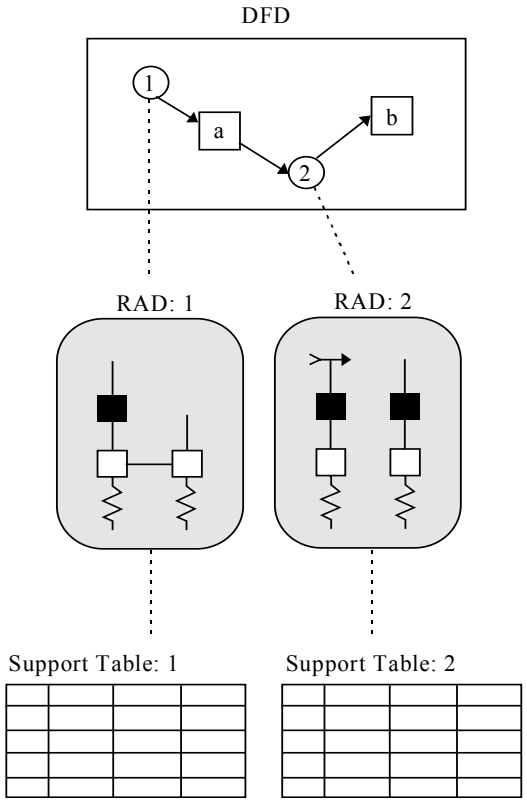



Figure H.2: Modelling Approach

**H.3.1.4** A data flow notation is used to present a high-level model of the whole development process; this provides an easy to understand entry point into the process definition.

**H.3.1.5** RADs are used to provide a more detailed model of individual processes and to capture process dynamics (eg iteration). The RADs refine the DFD by decomposing high-level development activities into a set of more detailed activities; an alternative approach would be to perform the decomposition using lower level DFDs.

**H.3.1.6** RADs do not easily capture the resources used by processes (eg tools and methods) and therefore each RAD has an associated support table to capture these aspects. The RADs map directly to the support tables via activities. Thus for each activity shown on a RAD there is corresponding support table entry.

**H.3.1.7** The data flow notation, RADs and the support tables will now be described in more detail together with examples from SHOLIS.

**H.3.1.8** It should be noted that the techniques used here are chosen for illustrative purposes only. Other techniques which meet the analysis objectives may be used instead.

**H.3.2** Data flow diagrams

**H.3.2.1** The first element of the SHOLIS process definition is a high-level DFD; see **figure H.3**. The data flow model was produced by consulting the various project plans and discussing the development process with project staff.

**H.3.2.2** The data flow notation used depicts development and checking activities as rounded boxes and software products as rectangles. Arrows indicate information flow.

**H.3.2.3** Only one DFD is drawn in order to provide a high level model of the entire process, and consequently not all activities are shown. For example the SHOLIS project has a number of separate safety analysis activities, eg FMEA and FTA, and these are shown as a single activity "safety analysis" on the DFD. The RAD associated with "safety analysis" then decomposes this activity into "produce FMEA", "produce FTA" and so on.

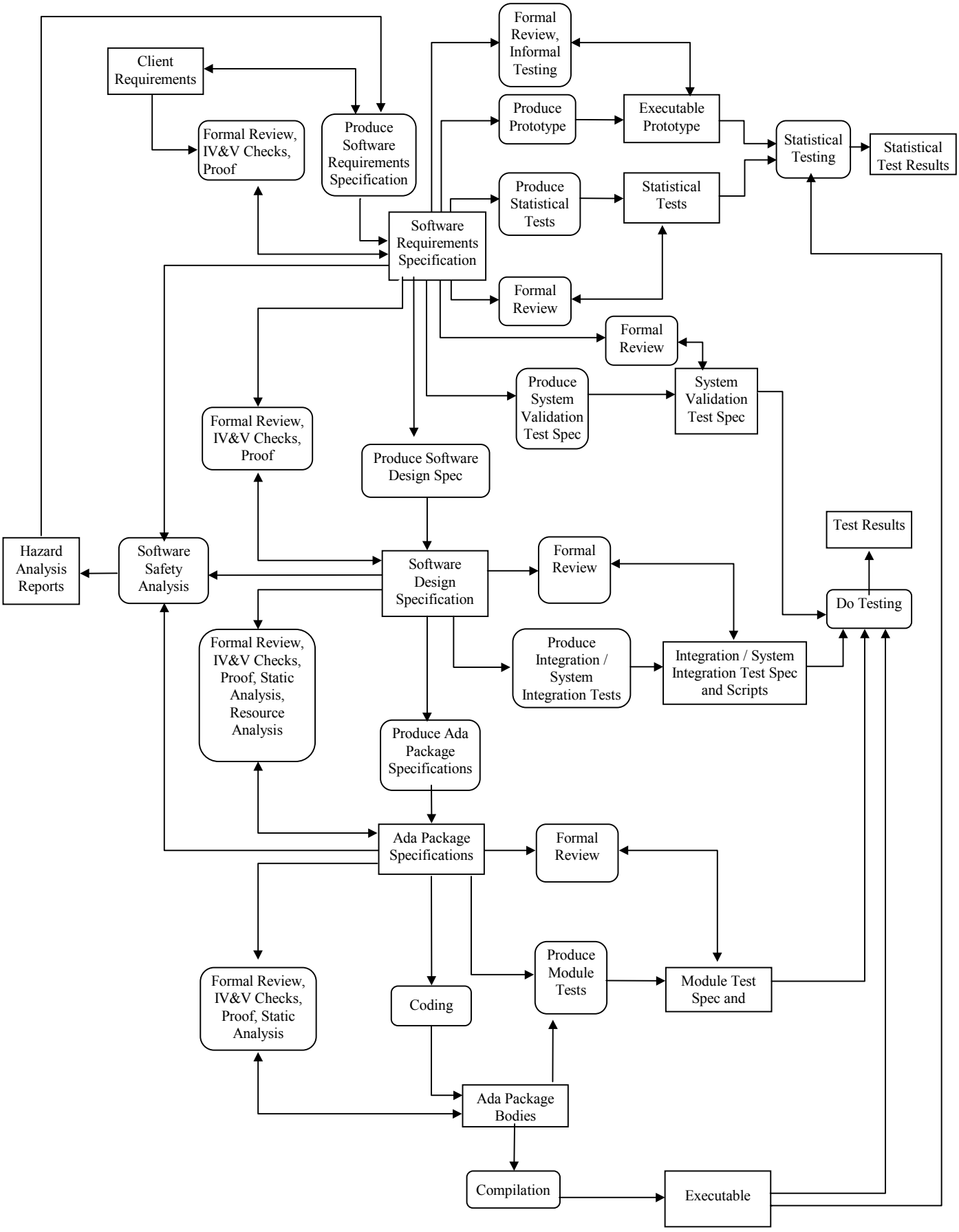



Figure H.3 : SHOLIS Development Process

**H.3.3** Role Activity Diagrams

**H.3.3.1** The high-level DFD is refined using a RAD to capture more detailed activities and process dynamics. An example of this modelling step is now given using the development activity "Produce Software Specification"; the RAD for this activity is shown in **figure H.4** and a key for the RAD symbols can be found in **table H.1**.

**H.3.3.2** For a process (at any given level of abstraction) a RAD shows the roles, their component activities and interactions together with external events and the logic that determines what activities are carried out and when.

**H.3.3.3** The Software Specification (SS) specifies the requirements for the SHOLIS system. The SS has the following components:

(a) Z specification of the software functionality;

(b) supporting English narrative;

(c) specification of the MMI.

**H.3.3.4** The RAD shows two roles involved in producing an issue of the SS: the Project Manager and the SS Team. The RAD shows the Project Manager's role as simply to assemble the team; this modelled as the interaction *SS team starts*. Obviously the Project Manager will monitor the progress of the SS Team on a regular basis, however, such interactions are considered outside the scope of this process model. The SS Team role can be viewed as having two main threads of activity: the new issue thread and the delta thread.

**H.3.3.5** The new issue thread involves those activities necessary to produce the next issue of the SS against some set of requirements (or review actions in the case where an issue is being produced as a result of a formal review). The RAD shows the new issue thread starting when the go-ahead is received by the SS Team to produce a new issue of the SS, this being modelled by the external event *New issue of SS required*. The exact nature of the go-ahead will vary depending on why the new issue is required, for example it could be the arrival of a new requirements document or as a result of a formal review.

**H.3.3.6** The delta thread involves those activities that handle asynchronous events which have an impact on the SS, for example a change request or a fault report; this thread is not considered further in the this document due to space limitations but would be fully analysed in practice.

**H.3.3.7** When the new issue thread starts, ie when a new issue of the SS is required, four parallel activities are shown as starting:

(a) Do tracing

(b) Do Z model

(c) Do narrative

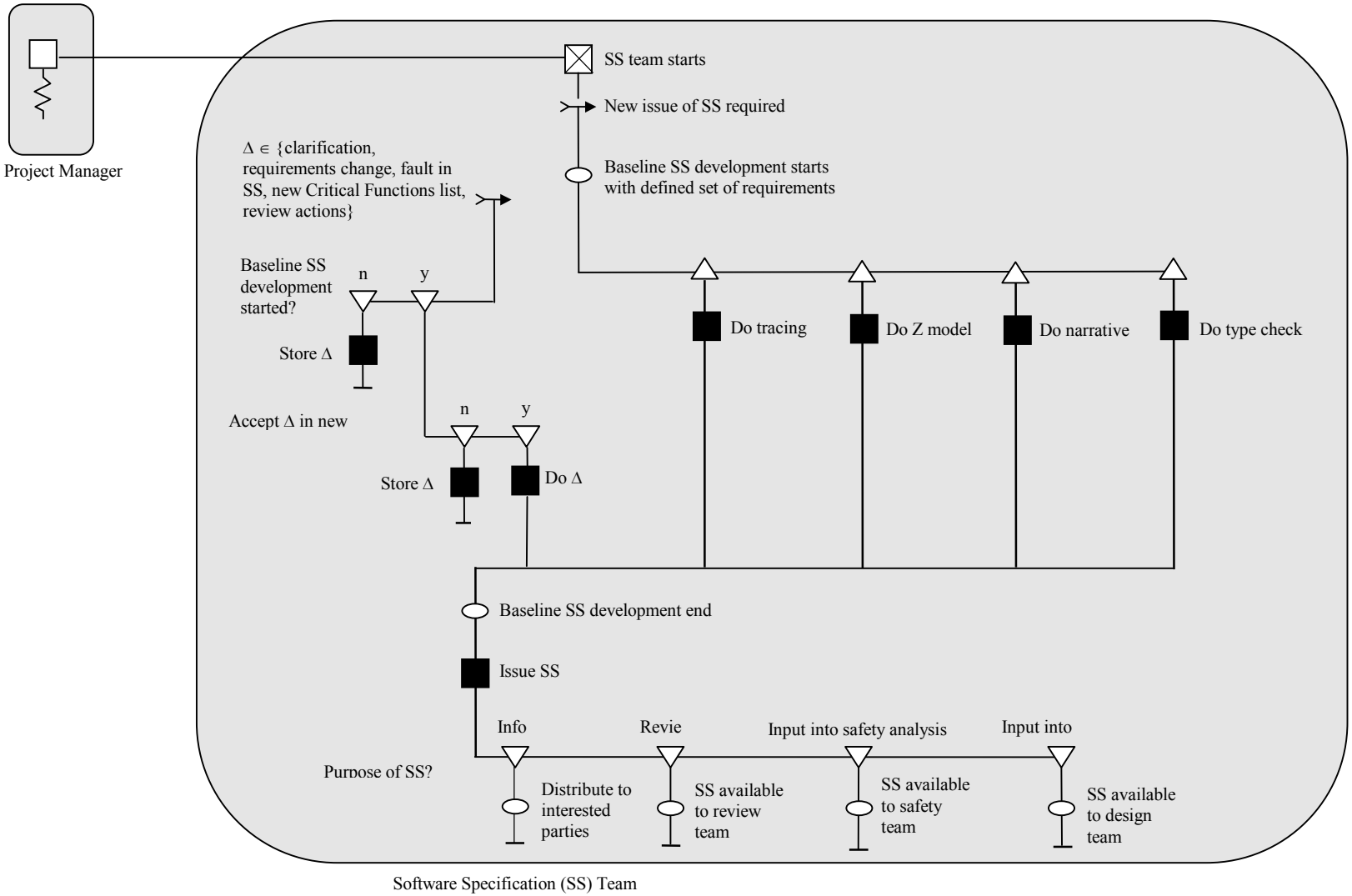
(d) Do type H check

**H.3.3.8** The RAD shows that eventually the tracing, Z, narrative and type checking activities complete for a given issue of the SS and it is issued (*Issue SS*) to relevant parties. Who makes up the "relevant parties" is shown by the RAD to depend on the *Purpose of the SS*. The SS may have more than one purpose, for example it could be distributed to the formal review team and to others for information only.

Table H.1 : RAD Symbol Key

| **Symbol** | **Meaning** |
|---|---|
|  | State description |
|  | Interaction starting a role |
|  | Activity |
|  | External event |
|  | Parallel activity |
|  | Decision |

Figure H.4: SHOLIS process model - RAD for Produce Software Specification



**H.3.4** Support Tables

**H.3.4.1** The RAD for "Produce Software Specification" identifies the more detailed development activities that take place in order to produce the specification. However, the RAD does not show any information regarding the resources that support these lower level activities. The support tables provide this information.

**H.3.4.2** One support table is required for each role identified on a given RAD. Therefore there are two support tables for "Produce Software Specification": one for the Project Manager role and one for the SS Team role.

**H.3.4.3** Within a support table there is one entry for each activity within a role. **Table H.2** illustrates what a support table entry details for a given activity.

Table H.2 : Support Table Contents

| **Support Table Entry** | **Description** |
|---|---|
| Activity | The name of the activity on the RAD to which the entry refers. |
| Inputs | The input to the activity, for example specifications or test results. |
| Tools | Computer-based tools to support the transition from the input to the output |
| Methods | Documented procedures/notations to support/enable the transition from input to output. |
| Checks | Fault detection measures outside of the activity. |
| Human Tasks | The role of human beings in supporting/enabling the transition from input to output. |
| Outputs | The outputs from the activity, for example a design, or executable code. |

**H.3.4.4** Extracts from the support table for the SS team role of the "Produce Software Specification" process are given in **table H.3**.

| **Role: Software Specification (SS) Team** | | | | | | |
|---|---|---|---|---|---|---|
| **Activity** | **Inputs** | **Tools** | **Methods** | **Checks** | **Human Tasks** | **Output** |
| **Do_tracing** | SSTR, SEA, SIP, OS (see **annex H** for abbreviations) | tracing tool in perl emacs | DCOP Chap 2 | formal review, IV&V checks, testing, safety analysis | extract requirements from inputs | SS: traceability matrix |
| **Do_Z_model** | SSTR, SEA, SIP, OS | emacs, LaTex | Z notation as defined in Spivey 2 (Z), DCOP Chap 4, Praxis Z standard, | formal review, output from Do_type_check, proof, testing, safety analysis | expression of requirements in Z | SS: Z model |
| **Do_narrative** | SSTR, SEA, SIP, OS, Z model | emacs, LaTex | DCOP Chap 4 | formal review, IV&V checks, testing, safety analysis, resource and timing modelling | express Z model in English, express aspects of system not captured in Z for example scheduling | SS: narrative explains Z model and describes requirements not specified in Z |
| **Do_type_check** | Z model in LaTex | fuZZ | fuZZ manual, DCOP Chap 4 | formal review, IV&V check, proof, testing | start type check, review results | SS: type check results |
| **Do_Δ** | Δ, Do_tracing, Do_Z_model, Do_narrative, Do_type_check | Do_tracing, Do_Z_model, Do_narrative, Do_type_check | Do_tracing, Do_Z_model, Do_narrative, Do_type_check | union of :Do_tracing, Do_Z_model, Do_narrative, Do_type_check | take account of Δ in all activities | SS: updated with Δ |
| **Issue_SS** | SS | doctools | CM Plan | | enter config info - including SS - into doc | baselined SS |

Table H.3 : Extract from: Support Table - Produce Software Specification - SS Team

## H.4 SHOLIS: Process Hazard Identification

**H.4.1** Having modelled the SHOLIS development process, the next stage is to identify the process hazards against which the hazard analysis should be performed.

**H.4.2** A subset of the process hazards identified for SHOLIS is listed in **table H.4**. These were identified using both a check list and the first level safety case claims - of which a subset are shown in **figure H.1**. For example, the first level claim '*safe functional behaviour exhibited*' was re-worded to become the process hazard '*hazardous functional fault in operational software*'.

**H.4.3** The completeness of the hazard list can be checked by ensuring that each and every software safety requirement maps to at least one of the process hazards. For example, consider the following SHOLIS safety requirements:

(a) Requirement 1: A roll alarm shall be raised if the input sensor value is valid and outside the roll limit, or if the input sensor value is invalid.

(b) Requirement 2: The emergency page shall be re-calculated every 3 minutes.

Requirement 1 is clearly functional in nature and is therefore covered by the process hazard **haz.func**. Requirement 2 is concerned with both functional and timing behaviour and is therefore covered by the process hazards **haz.func** and **haz.timing**. If a safety requirement was found not to be covered by any process hazard, then it must be concluded that the process hazard list was incomplete and corrective action must be taken.

Table H.4 : Process Hazards

| **Reference** | **Hazard Description** |
|---|---|
| **haz.func** | hazardous functional fault in operational software |
| **haz.timing** | hazardous timing fault in operational software |
| **haz.mem** | not enough memory available |

## H.5 SHOLIS: Process Failure Analysis

### H.5.1 Introduction

**H.5.1.1** Process failure analysis involves carrying out three related analyses: process failure mode analysis, cause-effect analysis and common mode failure analysis.

**H.5.1.2** A Failure Modes and Effects Analysis (FMEA) has been used to perform process failure mode analysis. The results of the FMEA are then summarised in a coverage matrix which shows which checks are applied to each development activity. The coverage matrix combined with the FMEA provide two contrasting levels of failure analysis. The matrix is an aid to the evaluation of the overall process.

**H.5.1.3** Cause-effect analysis is performed by taking the results of the FMEA and relating them to the various process hazards by means of Fault Tree Analysis (FTA). Fault trees allow an assessment of the causes of and defences against each process hazard, and they enable a comparison to be made between the level of defences provided for different process hazards.

**H.5.1.4** Common-mode failure (CMF) analysis has been performed by reviewing the SHOLIS FMEA and FTA, and producing tables showing the identified sources of CMF together with associated safety arguments.

**H.5.1.5** Each of these steps will now be described in more detail.

### H.5.2 Failure modes and effects analysis

**H.5.2.1** FMEA analysis traditionally considers component failures, and it is therefore necessary to have some notion of what constitutes a 'component' in the context of a software development. The main software development activities (eg produce software specification) will constitute the components for the purposes of this analysis.

**H.5.2.2** When considering software development activities it is useful to distinguish between:

(a) Development activities. These are activities that give rise to software products (ie objects that are, or transform into, the final product).

(b) Checking activities. These are activities that check the correctness and/or completeness of software products. Thus proof and unit testing are examples of checking activities. Additionally software safety analysis is considered a checking activity as it is essentially validating and verifying the safety requirements.

(c) Infrastructure activities. These support the various development and checking activities. Thus configuration management and fault management are examples of infrastructure activities.

**H.5.2.3** The FMEA components are therefore the development, check and infrastructure activities that form the development process. Identification of the activities fitting into the above categories can be done by examining the process definition; a subset of the SHOLIS development, check and infrastructure activities are listed in **table H.5**. The FMEA proceeds by completing a worksheet for each of the identified activities.

Table H.5 : SHOLIS activities

| **Activity Type** | **Activity** |
|---|---|
| **Development** | Produce software specification<br>Produce software design |
| **Checking** | Formal Review<br>IV&V miscellaneous checks (including tracing)<br>Proof (Z and SPADE)<br>Functional Failure Modes and Effects Analysis |
| **Infrastructure** | Configuration Management<br>Change Management |

**H.5.2.4** The FMEA components (ie activities) are functional entities and therefore the level at which they are defined can be determined by the analyst. Consequently the detail in the FMEA, and therefore the effort required to produce it, can be altered by changing the level at which the activities are defined. Diminishing returns will apply beyond a certain level of detail.

**H.5.2.5** The SHOLIS FMEA analysis is recorded on a series of worksheets, extracts of which can be found in **annex M**. The worksheets provide the following information:

(a) the activity name;

(b) the inputs and outputs of the activity;

(c) the resources that enable the activity to be performed, for example tools, methods, staff;

(d) the failure analysis for the activity, ie the FMEA proper.

**H.5.2.6** All but the failure analysis can be filled in directly from information contained within the SHOLIS process definition.

**H.5.2.7** The information provided by the failure analysis will now be considered in detail; the reader is advised to consult **annex M** whilst reading this annex.

**H.5.2.8** The first column of the failure analysis captures the activity failure modes. Development activities and infrastructure activities are characterised by the following generic failure modes:

(a) *Input fault not detected*. A development example would be a design activity not detecting an error in the input specification thereby leading to a defective design; thus the input fault is carried forward to the output.

(b) *Transformation fault*. Where a mistake, human or otherwise, results in an incorrect transformation from input to output. A development example would be a compiler error causing a discrepancy between the source and object code. An infrastructure example would be a change management system failing to correctly identify items affected by the proposed change.

**H.5.2.9** Checking activities have a single generic failure mode:

(a) *Input fault not reported*. That is, where a fault in the input is not detected and/or reported, for example where a type checker fails to report a type mismatch in a formal specification.

**H.5.2.10** An alternative approach to using the above generic failure modes would be to use a checklist of fault types, for example: incomplete and inconsistent syntax.

(a) The failure analysis then captures the causes of each of the generic failure modes. The approach used is to identify each of the resources that support a given activity and consider how each resource could fail so as to give rise the relevant generic failure modes.

(b) Identification of the causes of each generic failure is assisted by the use of HAZOPS-like guidewords which are applied to the purpose of each resource. The guidewords used together with example deviations are given in table H.6.

Table H.6 : Guidewords

| **Guideword** | **Example Deviation** |
|---|---|
| PART OF | Requirement partially modelled |
| AS WELL AS | Additional requirement modelled |
| OTHER THAN | Something completely different modelled. |
| NO/NOT | Requirement omitted. |

**H.5.2.11** The failure analysis continues by showing the end effect of the failure-cause combination. The end effects are one or more of the process hazards listed in **table H.4**. For example, the SHOLIS software specification models the software functional requirements in Z and other non-functional software requirements - such as timing - informally in the narrative. Resources associated only with the Z can only give rise to functional faults in the final delivered object code, ie they can only cause **haz.func**. In contrast, if the engineers writing the specification are responsible for modelling the Z and expressing timing requirements in the narrative, then they could cause both functional and timing faults in the final system, ie they could cause both **haz.func** and **haz.timing**.

**H.5.2.12** At this point in the failure analysis, failure modes and possible causes of them have been suggested for each of the identified 'components'. The next step of the failure analysis is to identify the defences against each failure-cause combination. Defences for development, checking and infrastructure activities fall into two categories:

(a) fault detection measures: actions taken to detect faults, for example proving a specification;

(b) fault avoidance measures: actions taken to avoid faults in the first place, for example using proven tools and competent staff.

**H.5.2.13** The final information captured by the failure analysis is to state or reference evidence to support the arguments made for the various defences. For example, a fault avoidance measure against human error may be the use of appropriate staff. Such a defence requires supporting evidence such as references to CVs and training records. Evidence is not referenced to support fault detection measures, since the safety argument for a given fault detection measure is already, or will be, provided the FMEA conducted on it. For example, if one of the defences against a specification error is 'Proof', then an examination of the FMEA conducted on the 'Proof' activity will show:

(a) the failure modes of the proof activity;

(b) the fault detection measures that are in place, for example the reviews carried out on the proof;

(c) the fault avoidance measures that have been taken, for example, the competence of the staff involved.

**H.5.2.14** Such information should enable an evaluation to be made as to how adequate a defence 'Proof' is likely to be against a specification error.

**H.5.2.15** Much of the evidence to support the various defences can be reused and therefore an Evidence Library has been used; this contains claims which are supported by direct or referenced evidence. The FMEA uses the Evidence Library by references of the type *evi_lib.xxxx* in the evidence column. An extract from the SHOLIS Evidence Library can be found in **annex K**.

**H.5.3** Coverage matrix. **Table H.7** provides a summary of the checks (fault detection) that are applied to each of the software products. When considering the table care should be taken to consider the relative strength and scope of the various checks: some checks may have very wide scope but have a low detection rate, eg formal reviews; others may have limited scope but a very high detection rate, eg automated syntax and type checking.

**H.5.4** Fault tree analysis

**H.5.4.1** Fault tree analysis can be used to make clear which process failures and which defences are relevant to the various process hazards. See **annex K** for an example.

**H.5.4.2** Developing a fault tree for each process hazard will enable a comparison to be made between the level of defences put in place to prevent them. So, for example, if a system had hard timing requirements then one would expect to see a similar level of defence against **haz.timing** as for **haz.func**; fault tree analysis enables such an assessment to be made relatively easily.

**H.5.5** Common mode failure analysis

**H.5.5.1** The defences against a given process hazard should be evaluated for their susceptibility to common mode failure (CMF).

**H.5.5.2** CMF analysis can be performed by a combination of some or all of:

(a) examining the results of failure analysis, for example FMEA;

(b) a brainstorming session to identify common points of failure, for example a HAZOP type study;

(c) reviewing the results of Fault Tree Analysis.

**H.5.5.3** **Table H.8** shows some of the common cause failures that affect the **func.safe_construction** and **func.safe_testing** claims. Note that references to the Evidence Library are used in the same way as in the FMEA.

Table H.7 : Coverage Matrix

| **Software Product** | **Fault Detection Measure** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | formal review (D) | IV&V misc checks (I) | syntax and type check (ID) | proof (ID) | static analysis (ID) | resource analysis (D) | unit testing (I) | integration testing (I) | system integration testing (I) | system validation testing (I) | acceptance testing (IC) | Statistical Testing (I) |
| software specification | ■ | ■ | ■ | ▨ | | | | | | | ■ | |
| software design specification | ■ | ■ | | ▨ | | | | ■ | | ■ | ■ | ▨ |
| Ada package specifications | ■ | ■ | ■ | ▨ | ■ | ■ | ■ | | ■ | ■ | ■ | ▨ |
| Ada package bodies | ■ | ■ | ■ | ▨ | ■ | | ■ | | ■ | ■ | ■ | ▨ |
| target object code | | | | | | | | | ■ | ■ | ■ | ▨ |
| executable | | | | | | | | | ■ | ■ | ■ | ▨ |

KEY

| | |
|---|---|
| | Check not applied to software product |
| ▨ | **Check only applied to safety attributes** |
| ■ | **Check applied to all (relevant) attributes** |

| | |
|---|---|
| I | **Performed/Checked by independent team** |
| D | **Performed/Checked by development team** |
| C | **Performed/Checked by Customer** |

Table H.8 : CMF Analysis Examples

| **Common Cause** | **Effect** | **Safety Argument** |
|---|---|---|
| The software requirements specification is the basis for both software development and system validation testing. | A fault in this specification (eg specification of the wrong requirements) would affect the validity of the results of both analysis and testing. | • Phase 1 of the SHOLIS project was a wide ranging and detailed requirements validation exercise. The assessment tests involved experienced Royal Navy personnel.<br><br>• The application domain of SHOLIS is well understood and manual systems have been in place over a long period. |
| All software development products are configured using the same systems. | Should an incorrect version of any configuration item be supplied by the configuration management system, the value of any analysis or tests on that item are undermined. | • evi_lib.config_man |

Collation Page

Process Evidence

## J.1 Introduction

**J.1.1** Part 1 of this Standard calls for a justification of the software development process. This is achieved, in part, by performing a safety analysis on the software development process and this is covered in **.4**.

**J.1.2** During process safety analysis various detailed claims will be made about the software development process and these claims will need to be supported by evidence. This appendix discusses the different forms of process evidence that can be used to support such claims.

**J.1.3** Guidance on the collection and presentation of process evidence is split into three topics:

(a) Staff Competency - Evidence should be supplied that all staff working on developing and maintaining SRS are competent and have the appropriate skills, qualifications, experience and training.

(b) Methods - The suitability of the specification, design, coding and testing methods. Evidence might include successful use on previous projects.

(c)Tools - The fitness for purpose of the tools used on the project. Particular attention should be paid to compilers and configuration management tools. Evidence may take the form of tool evaluations possibly using established assessment suites.

## J.2 Staff Competency

**J.2.1** Staff competency should be demonstrated to ensure that all staff developing and maintaining SRS have an appropriate level of relevant experience, are properly trained, suitably qualified and have the necessary skills to carry out their tasks on the project.

**J.2.2** An appraisal of staff competency should be carried out and included in the Software Safety Case. Evidence should show that individuals do have the necessary attributes described above. Particular attention should be paid to assessing and providing evidence about the key members of the software development team.

**J.2.3** Pending the results of the Health and Safety Executive sponsored Competencies Study, the British Computer Society Industry Structure Model (ISM) may be a useful guide. The ISM is a set of performance standards designed to cover all functional areas of work carried out by practitioners and professionals in the computer industry. It provides a set of guidelines to define training needs for career development and a set of performance criteria against which experience and training can be assessed and is intended for use by both employers and employees.

**J.2.4** Evidence should be collected for each member of the project team of their role, grade and a summary of their training, qualifications and experience specifically relevant to their role on the project. Up-to-date CVs should be maintained.

**J.2.5** Details of project specific training that is either planned or has been carried out should be included as evidence. Changes of any key member of staff during a project should also be recorded and a justification provided to show that the effects of the change have been considered and appropriate action (eg a suitable handover period) taken.

**J.3** Methods

**J.3.1** Evidence is required to justify the use of the software development methods used to develop the SRS. The following should be considered during the process of selecting a method:

(a) compliance with the requirements of this Standard;

(b) the cost and availability of tool support;

(c) the availability of training and expert help in using the method (if necessary);

(d) the amount and depth of experience that the software development team have in the method;

(e) the readability and understandability of the notations used by the method;

(f) the maturity of the method (eg commonly practised, becoming increasingly practised, rarely practised, not practised);

(g) whether the method has been used successfully by the supplier or others in industry for similar systems of at least the same integrity level;

(h) the power of the method (in terms of its ability to express, model, check or analyse the problem at hand);

(i) the compatibility between the method and the other methods and tools used by the project.

**J.3.2** Information on the above aspects should be used as evidence in the Software Safety Case to support the justification of the methods used.

**J.4** Tools

**J.4.1** Part 1 of this Standard requires a justification of the tools used to develop the SRS with respect to the effects of possible faults in them on the safety integrity of the SRS.

**J.4.2** It is important to provide evidence for transformational (eg compilers, linkers, optimises and loaders) and infrastructure tools (eg configuration management tools) as failures of these tools may introduce faults directly into the software.

**J.4.3** The types of evidence that should be collected about a tool are:

(a) the results of evaluation tests or trials performed on the chosen tool and other candidate tools, if any. The evidence should contain a list of the criteria used in the tests/trials; detailed results of any trials carried out and the justification for the choice of a particular tool;

(b) the results of assessments of the quality of the chosen tool. The assessment evidence should contain a summary of the quality standards to which the tool was developed; detailed results of quality audits carried out on the tool supplier (including any specific quality standards);

(c) an analysis of the possible faults in the tools and their effects on safety plus a summary of any measures taken to reduce or remove the effects of these faults;

(d) a list of faults in the chosen tool, whether these faults have been fixed and, if not, how the effects of the faults have been avoided eg avoiding certain language constructs;

(e) experience of using the tool on previous projects.

**J.4.4** The guidance and requirements of RTCA/DO-178B with respect to Tool Qualification may be found useful as an approach to gathering the necessary evidence.

**J.4.5** Particular attention should be paid to the assessment and evaluation of the compilation system. Consideration should be given to evaluating the compiler by using:

(a) compiler evaluation test suites;

(b) commercially available compiler test tools.

**J.4.6** The run-time system on which the program will execute on the target computer should also be subject to evaluation. Some compiler vendors provide special versions of run-time systems which are certified or allow third-party evaluation to be undertaken.

Collation Page

SHOLIS Evidence Library

**K.1** The purpose of the Evidence Library is to develop reusable claims regarding the suitability of methods, tools and staff. These claims are supported by evidence which is presented directly or referenced. **Table K.1** contains a sample of the claims referenced from the SHOLIS FMEA worksheets.

Table K.1 : Extract from SHOLIS Evidence Library

| Evidence Ref | Applies to | Claim | Evidence |
|---|---|---|---|
| evi_lib.z | Method: Z | Z is a mature and sound notation suitable for the formal specification of complex systems. | The formal semantics for Z have been partially defined by Spivey (1987) and the Z ISO standard working group are currently defining the full formal semantics of the language.<br>Z can be considered proven-in-use having been used on a large number of industrial and research projects.<br>Z has been used on many Praxis projects including : 204, 213, 243, 307, 343, 352, 364 |
| evi_lib.emacs | Tool: emacs | emacs is a suitable general text editor for use on projects developing safety-related and safety-critical systems. | Emacs is widely used within Praxis and is considered as robust as any comparable text editor. It is the most widely used text editor on Unix platforms and has been widely ported to others. Emacs is well documented, see manual in SHOLIS project file. |
| evi_lib.fuzz | Tool: *f*uZZ | *f*uZZ is a mature tool and highly effective at the detection of syntax and typing errors in Z. The reports produced by the tool are easy to understand and unambiguous. | Tool produced by J.M.Spivey author of the defacto Z reference manual.<br>The tool has been in use since 1988 and has a medium sized user base. The tool is used for the syntax and type checking of all Z formal specifications in Praxis.<br>The tool has been used on many Praxis projects including: 204, 213, 243, 307, 339, 343, 352 and 364.<br>All error messages produced by the tool are listed in the *f*uZZ manual together with reference to the relevant language rule (as defined in Spivey2) that has been broken. |
| evi_lib.staff.ano | Staff: A.N.Other | A.N.Other possesses the appropriate skills and experience to specify and prove complex systems formally. | A.N.Other is an experienced formal methods practitioner with experience of writing formal specifications in Z for defence, avionics and rail projects.<br>A.N.Other has a BSc(Hons) in Computation and Mathematics and is a member of the Z standards working group.<br>A.N.Other is a BCS Level 6 practitioner.<br>For more details consult the relevant CV in the SHOLIS project file. |

Fault Tree Analysis

**L.1** Part of the fault tree analysis for process hazard **haz.func** is now given.

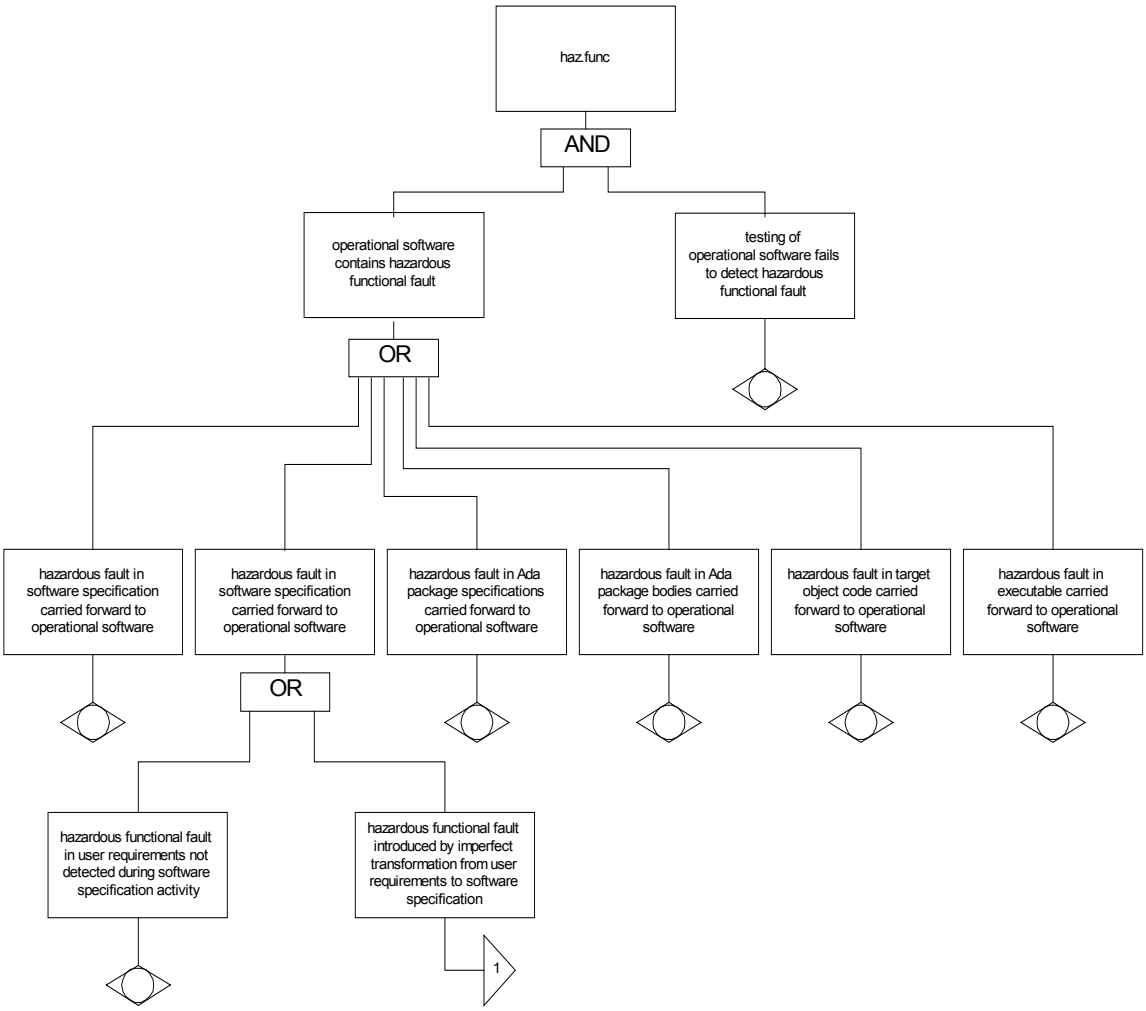



Figure L.1 Fault Tree Analysis for **haz.func** (1)

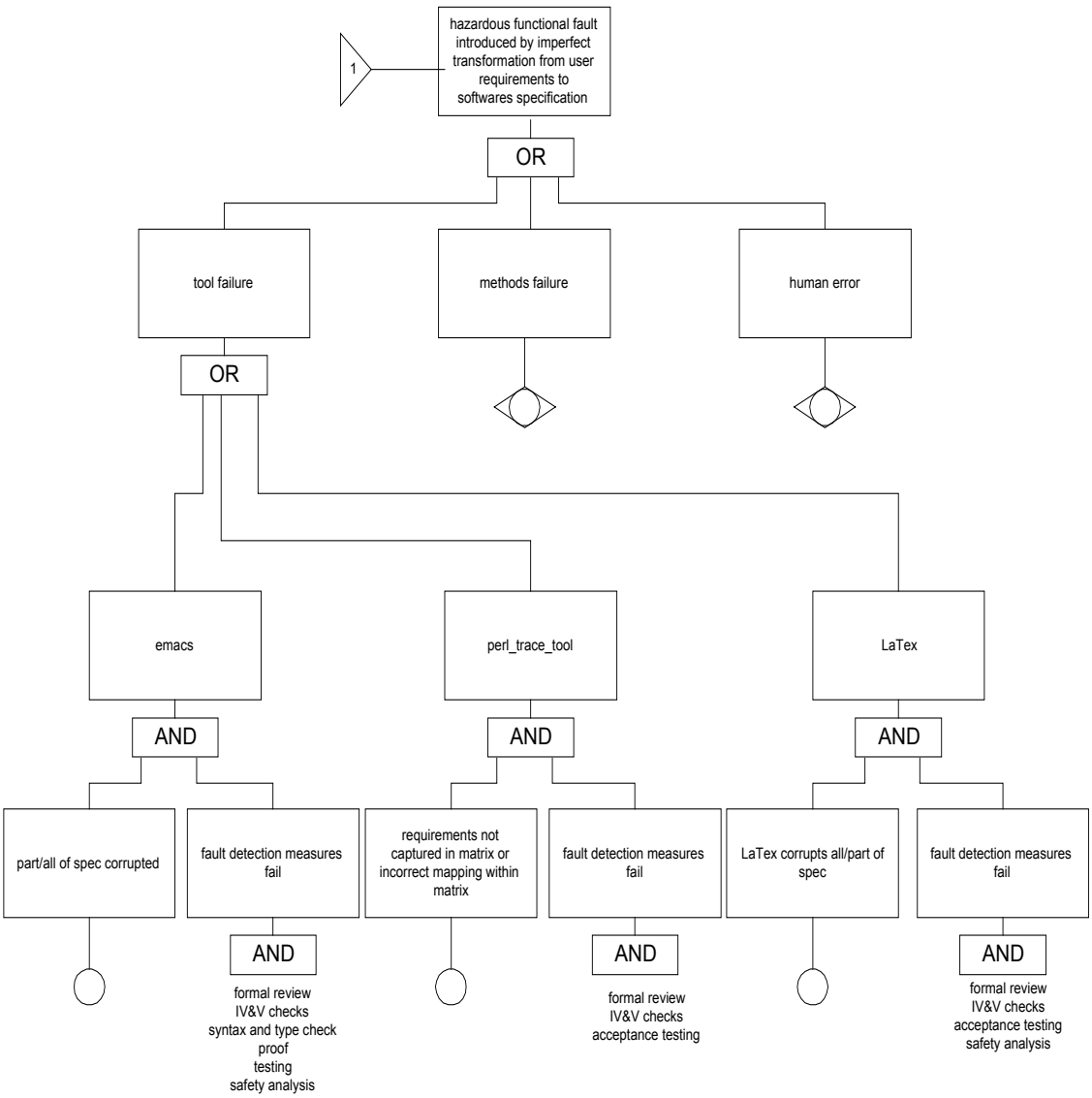



Figure L.2 Fault Tree Analysis for **haz.func** (2)

SHOLIS: FMEA Worksheets

## **M.1** Introduction

The SHOLIS FMEA worksheets are divided into two sub-sheets. The first sheet, the header, details activity information such as inputs, outputs and resources used. The second sheet, the body, details the actual failure analysis.
The distinction between fault avoidance and detection measures is shown as follows:

■ - fault avoidance - fault detection

Table M.1: FMEA Worksheet Extract: Produce Software Specification Header

| SHOLIS FMEA Worksheet - HEADER | Activity: Produce Software Specification | Type: Development | |
|---|---|---|---|
| Inputs | Statement of System Technical Requirements (SSTR)<br>System Elements Analysis (SEA)<br>System Implementation Plan (SIP)<br>Operational Specification (OS)<br>Software Hazard Analysis Report | Output | Software Specification |
| Resources | | | |
| Tools | Methods | Checks | Human Role |
| emacs<br>LaTex<br><br>perl_trace_tool | Design Codes of Practice<br>Z notation | formal review<br>IV&V misc checks<br><br>syntax and type check<br><br>proof<br>acceptance testing<br>back-to-back testing<br>safety analysis | Extract requirements from inputs.<br>Express functional requirements in Z<br>Express non-functional requirements<br>Explain Z model in English |

Table M.2 FMEA Worksheet Extract: Produce Software Specification

| SHOLIS FMEA Worksheet - BODY | | | Activity: Produce Software Specification | | Type: Development |
|---|---|---|---|---|---|
| Ref | Failure Mode | Cause | End Effect | Defences/Measures<br>■ - fault avoidance - fault detection | Evidence |
| **0005** | **Input fault not detected.** | **Method: DCOP: inadequate formalization of non-functional requirements results in faults in non-functional requirements not being detected.** | **haz.timing**<br>**haz.mem**<br>**haz.op** | 1 formal review<br>2 IV&V checks<br>3 acceptance testing<br>4■ close mapping from Z to MMI description<br>5■ close mapping from Z to performance reqs<br>6 safety analysis | **4■ see DCOP section 4**<br>**5■ see DCOP section 4** |
| **0010** | | **Method: DCOP: inadequate formalization of functional requirements results in fault in requirements not being detected.** | **haz.func** | 1 formal review<br>2 IV&V checks<br>3 acceptance testing<br>4■ modelled in Z<br>5 safety analysis | **4■ evi_lib.Z** |
| **0015** | | **Method: Z: inappropriate specification language for the application leading to faults in the requirements being missed.** | **haz.func** | 1 formal review<br>2 IV&V checks<br>3 acceptance testing<br>4■ Z appropriate for SHOLIS type system<br>5 safety analysis | **4■ evi_lib.Z.system_types** |

Table M.2 (Cont)

| Ref | Failure Mode | Cause | End Effect | Defences/Measures<br>■ - fault avoidance - fault detection | Evidence |
|---|---|---|---|---|---|
| **0020** | | **Human: engineer fails to spot inconsistency/ambiguity/incomplete nature of requirements.** | **haz.all** | **1 formal review**<br>**2 IV&V checks**<br>**3 phase I assessment testing**<br>**4 acceptance testing**<br>**5■ expert author**<br>**6■ formalization, ie modelled in Z**<br>**7 safety analysis** | **5■ evi_lib.ano**<br>**6■ evi_lib.Z** |
| **0025** | **Imperfect transformation from input to output.** | **Tool: emacs: corrupts part/all of specification** | **haz.all** | 1 formal review<br>2 IV&V checks<br>3 syntax & type check<br>4 proof<br>5 acceptance testing<br>6■ emacs considered proven-in-use<br>7 safety analysis | **6■ evi_lib.emacs** |
| **0030** | | **Tool: LaTex: corrupts part/all of specification** | **haz.all** | **1 to 5 as above**<br>**6■ LaTex considered proven-in-use**<br>**7 safety analysis** | **6■ evi_lib.latex** |
| **0035** | | **Tool: perl_trace_tool: requirements not captured in matrix, incorrect mapping in matrix** | **haz.all** | **1 IV&V checks**<br>**2 formal review**<br>**3 acceptance testing** | |

| Ref | Failure Mode | Cause | End Effect | Defences/Measures<br>■ - fault avoidance - fault detection | Evidence |
|---|---|---|---|---|---|
| **0040** | | **Method: DCOP: inadequate checks on requirements coverage** | **haz.all** | **1 IV&V checks**<br>**2 formal review**<br>**3 acceptance testing**<br>**4■ SS review checklist includes checks on: reqs coverage, justification of omissions and correct labelling of trace units** | **4■ see DCOP appendix A** |
| **0045** | | **Method: DCOP: inadequate formalization of non-functional requirement** | **haz.timing haz.mem haz.op haz.maintain** | **see 0005** | |
| **0050** | | **Method: DCOP: inadequate formalization of functional requirements** | **haz.func** | **see 0010** | |
| **0055** | | **Method: Z: notation flawed leading to ambiguous model** | **haz.func** | **1 formal review**<br>**2 IV&V checks**<br>**3 proof**<br>**4 acceptance testing**<br>**5■ Z considered sound**<br>**6 safety analysis** | **5■ evi_lib.Z** |

Table M-4 (Cont)

| Ref | Failure Mode | Cause | End Effect | Defences/Measures<br>■ - fault avoidance - fault detection | Evidence |
|---|---|---|---|---|---|
| **0060** | | **Method: Z: inappropriate notation for application leading to difficult to model requirements.** | **haz.func** | **1 to 4 as above**<br>**5■ Z appropriate for SHOLIS type system**<br>**6 safety analysis** | **5■ evi_lib.Z.system_type** |
| **0065** | | **human: engineer fails to model requirement or models additional requirement** | **haz.all** | 1 formal review<br>2 IV&V checks<br>3 acceptance testing<br>4■ use of perl_trace_tool<br>5■ expert author<br>6 safety analysis | **4■ TBD**<br>**5■ evi_lib.ano** |
| **0070** | | **human: engineer specifies non-functional requirements incorrectly** | **haz.timing haz.mem haz.op haz.maintain** | **1 formal review**<br>**2 IV&V checks**<br>**3 acceptance testing**<br>**4■ expert author**<br>**5 safety analysis** | **4■ evi_lib.ano** |
| **0085** | | **human: engineer specifies functional requirements incorrectly** | **haz.func** | 1 to 4 as above plus<br>5 syntax and type check<br>6 proof<br>7 safety analysis | |

**Error! AutoText entry not defined.**Abbreviations

| | | |
|---|---|---|
| **N.1** | ALARP | As Low As Reasonably Practicable |
| **N.2** | CMF | Common mode failure |
| **N.3** | COTS | Commercial Off-The-Shelf |
| **N.4** | ISM | Industry Structure Model |
| **N.5** | IV&V | Independent V&V |
| **N.6** | MOD | Ministry of Defence |
| **N.7** | RAD | Role Activity Diagrams |
| **N.8** | RTCA | Radio Technical Commission for Aeronautics |
| **N.9** | SRS | Safety-related software |
| **N.10** | V&V | Verification and Validation |

# Defence Standard 00-55 (Part 2)/2

# Requirements for Safety Related Software in Defence Equipment

## Annexes E through M

## Guidance on the Development of a Software Safety Case

*Note: This document has not passed through a full consultative process. It provides guidance on a particular approach to the development of a Software Safety Case which satisfies the requirements of Part 1 of this Standard.*
*Any comments on its contents or recommendations for improvement should be sent to Stan Ops 2, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow, G2 8EX.*

Collation Page

DEF STAN 00-55 (PART 2)/2



Published by and obtainable from:
Ministry of Defence
Directorate of Standardization
Kentigern House
65 Brown Street
GLASGOW G2 8EX

Tel No: 0141 224 2531



The following Defence Standard file reference relates to the work on this Standard - D/D Stan/303/12/3

Contract Requirements

When Defence Standards are incorporated into contracts users are responsible for their correct application and for complying with contract requirements.

Revision of Defence Standards

Defence Standards are revised when necessary by the issue either of amendments or of revised editions. It is important that users of Defence Standards should ascertain that they are in possession of the latest amendments or editions. Information on all Defence Standards is contained in Def Stan 00-00 (Part 3) Section 4, Index of Standards for Defence Procurement - Defence Standards Index published annually and supplemented periodically by Standards in Defence News. Any person who, when making use of a Defence Standard encounters an inaccuracy or ambiguity is requested to notify the Directorate of Standardization without delay in order that the matter may be investigated and appropriate action taken.