CHAPTER 51

SOFTWARE RELIABILITY TECHNIQUES

CONTENTS

1 INTRODUCTION

1.1 This chapter contains guidance on the two complementary approaches to the achievement of software reliability at the design and implementation phase:

a) **Fault Avoidance**. This requires taking steps to avoid faults during software development, and to detect and correct those faults that do occur.

b) **Fault Tolerance**. It is unlikely that fault avoidance will succeed completely, and to achieve high reliability it is also necessary to design the software to correct or tolerate errors in service.

1.2 More information on the methods and techniques discussed in this chapter can be obtained from the publications listed in Leaflet 51/0.

2 FAULT AVOIDANCE

2.1 The following techniques should be considered when developing a fault avoidance strategy, each of which are described in more detail in the following paragraphs:

a) Inspections;
b) Classification of Requirements;
c) Checklists;
d) Traceability Analysis;
e) Requirements Inspection;
f) Specification Notations;
g) Structured Design;
h) Prototyping;
i) Formal Methods;
j) Reliability Analysis;
k) Static Analysis;
l) High Level Programming Languages;
m) Measures (Metrics);
n) Subjective Requirements Evaluation;
o) Testing;
p) Human Factors in Software Design;
q) Design Maturity; and
r) Analytical Arguments.

2.2 Many of these methods provide data that can be used to assess reliability achievement and to improve the software engineering process. Data collection and analysis schemes are described in BS 5760 Part 8[1].

2.3 Inspections

2.3.1 Inspections should be carried out on all the intellectual products of the software development life-cycle, including specifications, designs, code, test plans and test results. The objective of such inspections is to detect errors and to ensure that the item under review conforms to higher-level specifications where applicable. They should review:

a) The correct implementation of the component's specification, and the traceability of this to the system requirements;

b) The apportionment of reliability allocation;

c) Violation of standards and codes of practice;

d) Inspections may be carried out by means of;

e) A desk check by an independent reviewer; and

f) A walk-through (one type of walk-through is the Fagan inspection).

2.3.2 It may be helpful to use a simple checklist to guide the inspection.

2.4 Classification of Requirements

2.4.1 The software requirements should be classified and organized to promote comprehension and expedite subsequent analyses. The classification should:

a) Explicitly classify the requirements according to their object, e.g. whether they apply to the software product, the software engineering process, human-computer interaction, or the standards to be applied;

b) Identify requirements on software product design (e.g. defensive programming or software architecture);

c) Identify if there are any requirements for Built In Test (BIT) features;

d) Indicate the reliability level of each requirement;

e) Identify those requirements that are likely to change over the system's lifetime; and

f) Analyse the text to identify and clarify the use of specialist notations, terms, acronyms, and non-standard usage of common vocabulary.

2.5 Checklists

2.5.1 Checklists should be developed for reviewing the completeness and correctness of the requirements. Checklists should be based on data from previous projects.

2.5.2 One possible checklist is given below:

Interfaces

a) Are failure modes known and their detection and handling specified?

b) Is the software's response to out-of-range values specified for every input?

c) Is the software's response to not receiving an expected input specified? Does the specification define the length of the time-out, when to start counting the time-out, and the latency of the time-out (i.e. the point past which the receipt of new inputs cannot change the output result, even if they arrive before the actual output)?

d) Is a response specified if the input arrives when it should not?

e) On a given input, will the software always follow the same path through the code?

f) Is each input bounded in time? That is, does the specification include the earliest time at which the input will be accepted and the latest time at which the data will be considered valid?

g) Are the minimum and maximum arrival rates specified for each input and communication path? Are checks performed in the software to avoid signal saturation? Is the response defined if saturation occurs?

h) If interrupts are masked or disabled, can events be lost?

i) Can any output be produced faster than it can be used (absorbed) by the interfacing module? Is overload behaviour specified?

j) Are all data output to the buses from the sensors used by the software?

k) Can input that is received before start-up, while off-line or after shutdown influence the software's start-up behaviour? Is the earliest or most recent value used?

Robustness

a) In cases where performance degradation is required by the system specification as a means of fault handling, is the degradation predictable?

b) Are there sufficient delays incorporated into the error-recovery responses?

c) Are feedback loops specified, where appropriate, to compare the actual effects of outputs on the system with the predicted effects?

d) Are all modes and modules of the specified software reachable?

e) If hazard analysis has been done, does every path from a hazardous state lead to a low-risk state?

f) Is the receipt verified of the inputs that, if not received, can lead to a hazardous state or can prevent recovery?

Data consistency

a) Are checks for consistent data performed before control decisions are made based on that data?

b) If diverse or redundant hardware or software is used, is a vote taken before any key decision where data may differ between channels?

c) If diverse or redundant hardware or software is used, is a vote taken to ensure that remembered values (i.e. the internal state) are consistent between channels?

2.6 Traceability Analysis

2.6.1 Each requirement in the software requirements, including derived requirements, should be traced to the corresponding requirement in the system specification. A compliance matrix should be provided that summarizes how each requirement in the system specification and supporting documentation (e.g. standards) is to be implemented.

2.6.2 The dependencies between requirements should be analysed and documented.

2.6.3 The rationale for each requirement should be recorded within the limits imposed by security considerations.

2.7 Requirements Inspections

2.7.1 A formal inspection of the software requirements should be carried out. This should check that the following are adequately specified:

a) Software reliability requirements;

b) Functional behaviour, distinguishing any critical functions;

c) Capacity and response time performance;

d) Configuration or architecture of the overall system as far as this affects the software;

e) All interfaces between the software and other equipment or operators;

f) All modes of operation of the system in which the software is required to operate, including any fail soft modes;

g) Measures to overcome failure modes of the system, hardware or software (i.e. fault detection and fault tolerant mechanisms) that are to be implemented in software;

h) Requirements for software self-monitoring and BIT features;

i) Areas of functionality which are likely to change; and

j) Background information to enable the Reliability Case to summarize the system level design approach to reliability.

2.7.2 General guidance on inspections is given in paragraph 2.3.

2.8 Specification Notations

2.8.1 The use of a notation designed to support requirements specification (e.g. UML) should be considered. Such notations contribute to reliability by avoiding the ambiguity and inconsistency of natural language, and also allowing better structuring and traceability of requirements.

2.8.2 When choosing a special specification notation, it is important to consider how easy it will be for other parties to understand it. Problems can be avoided by:

a) Choosing a notation in which expertise exists in the interested parties' organisation;

b) Making provision for training other parties in the notation to an appropriate level; and

c) Providing a commentary on the requirements specification in English.

2.9 Structured Design

2.9.1 Structured design methods provide a methodical approach to software design by providing a set of notations and guidelines. Because they involve the production of a large number of design diagrams, tool support is essential. Examples are Structured Design (Yourdon), Jackson System Development (JSD) and MASCOT.

2.10 Prototyping

2.10.1 One way of reducing specification errors is by producing a prototype of the software, early in the life-cycle, for the users to experiment with. It may be better to produce several prototypes to examine different aspects of the system (e.g. one prototype for basic functionality, one for user interface, etc.). Generally, prototypes will not attempt to meet all requirements of the final system.

2.10.2 In order to be useful, prototypes have to be produced quickly, often using a different language from the final implementation. Thus a prototype is unlikely to meet all the reliability requirements for the system and should be discarded when prototyping is completed. The discard policy should be stated in the R&M Plan.

2.10.3 Specification Animation. Specification animation is a form of prototyping carried out directly from the notation used for the specification. For example, some formal methods tools allow specifications to be directly executed. The objective of specification animation is to confirm that the specification captures the user's requirements.

2.10.4 As with any prototype, a specification animation will probably be deficient in areas such as response time. Any areas that cannot be explored during animation should be recorded.

2.10.5 The specification animation should have easily used interfaces to encourage the purchaser's and user's representatives to explore the functionality of the specification. However, specification animations should also be tested, using formally recorded test cases and results. These tests should be designed to meet specific criteria that have been planned in advance.

2.10.6 A specification animation is one method of providing a diverse implementation to check the results of statistical testing (see PtCCh52 paragraph 2.2).

2.11 Formal Methods

2.11.1 Formal methods are specification and development methods for software and hardware that have a mathematical basis. They can be used as specification and design notations, and also as a means of carrying out Verification and Validation (V&V) against the module specifications. This application, which is variously known as program proof, verification condition generation and discharge, semantic analysis, and compliance analysis, is supported by several tools. Formal methods may also be cost-effective for complex real-time and concurrent systems, which are often impossible to reason about informally.

2.11.2 Formal methods require highly-trained staff and are best applied selectively (i.e. to relatively small, critical systems) and assist in the achievement of software reliability in several ways:

a) They provide a precise and unambiguous way of representing software and force the specifier to address the details;

b) They can be reasoned about mathematically, which enables them to be verified and validated much more thoroughly than is possible with an informal specification;

c) They can often be executed directly, or after a single design step, which makes it easy to carry out prototyping;

d) They provide a way of constructing analytical arguments.

2.12 Reliability Analysis

2.12.1 Software failures should be considered during system-level Failure Modes and Effects Analysis (FMEA) and Failure Modes, Effects and Criticality Analysis (FMECA) carried out as described in PtCCh33. At the early stages of the design, the software may be considered as a single entity; as the design progresses, it may be beneficial to carry out more detailed analyses on software subsystems or modules, particularly for complex or critical software.

2.12.2 For complex or critical software, detailed reliability analysis may be carried out by means of techniques such as FMEA, FMECA, software Fault Tree Analysis (FTA) or software hazard and operability studies. Analyses should be carried out by reference to the software requirements, which should be prepared from the system specification.

2.12.3 The results of software reliability analysis should be used as inputs to the system and software design processes to consider if the software failures are acceptable and, if not, devise appropriate fault avoidance or fault tolerant strategies.

2.13 Static Analysis

2.13.1 Although "static analysis" can apply to any V&V technique that does not involve executing the software, the term is particularly used for tool-supported V&V that investigates the software source text for control flow and data use anomalies. The sorts of static analysis that can be carried out include:

a) **Subset Analysis** - identification of whether the source code complies with a defined subset (see paragraph 2.14.4);

b) **Metrics Analysis** - evaluation of defined code measures, for example relating to the complexity of the code, often against a defined limit (see paragraph 2.15);

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** - the comparison of the code with a mathematical expression of the relationship between inputs and outputs. Semantic analysis can also be used to carry out checks against language limits, for example array index bounds or overflow. Semantic analysis is an example at the code level of the use of formal methods (see paragraph 2.11);

g) **Performance Analysis** - analysis of worst case conditions for any non-functional performance attributes, including timing, accuracy and capacity;

h) **Compliance Analysis** – a formal verification technique of checking conformance of the source code to the software specification. It enables assertions about the functional behaviour of the code to be proved correct.

2.14 High Level Programming Languages

2.14.1 The high level language used should be appropriate to the problem domain. In general, the fewest coding errors will occur if a strongly typed, highly structured language such as Modula-2 or Ada is used. Assembler should be restricted to the following:

a) Sections of the software where close interaction with hardware is required that cannot easily be accommodated with a high level language;

b) Situations where performance constraints cannot be met by a high level language; or

c) Very small applications where the use of a high level language, compiler and more powerful processor would increase, rather than reduce, the likelihood of faults.

2.14.2 Assembler code will require additional verification and validation.

2.14.3 Object Oriented Methods. Object oriented methods support design based on information hiding. Objects communicate by means of messages rather than by sharing data. It is good practice to ensure that objects are independent, and the representation of the state and operations on the state within an object are hidden from other objects. The use of object oriented methods should improve maintainability because changes to the internal representations of a module can be made without affecting other modules. The independence of modules also facilitates reuse. However, developing an object oriented view of an essentially functional design can be difficult.

2.14.4 Language Subsets. For critical software, the use of a high integrity language subset could be considered. Such subsets remove constructs from the full language that are difficult to analyse or prone to lead to programmer error. Commercially available subsets exist for Ada and Pascal, and guidance on sub setting is available for C and C++.

2.15 Measures (Metrics)

2.15.1 Statistical monitoring of the software engineering process (in an analogous way to hardware manufacturing) by recording some *measures* (metrics) of the process itself and about the resulting software product should be considered. These measures are best used comparatively, for instance to compare project progress to previous, similar projects, or to

identify error-prone modules within a software system. Some appropriate measures are described below.

2.15.2 Fault Density Measures. By far the most common quality measure is the number of residual faults per thousand lines of code (kloc), although this figure depends on the interpretations of "line of code" and "fault", and the point in the life-cycle at which it is measured. This measure can also be applied to specification and design documents (e.g. specification errors per thousand words).

2.15.3 In general, the most useful role for fault density measures is as a general indicator of the quality of the software engineering process. However, recent research has shown that the total number of faults in the software can be used to place a conservative bound on reliability growth in service (see PtCCh52 paragraph 4)**.**

2.15.4 Process Measures. Process measures are measurements of the software engineering process rather than of the software itself. An example is the five maturity levels defined by the Software Engineering Institute (SEI) at Carnegie Mellon University in work sponsored by the US Department of Defense. Contractors may wish to carry out exercises themselves (or use external contractors) to evaluate their processes against these levels, and endeavour to move to the higher levels.

2.15.5 Static Measures. Static measures are measurements of the size or complexity of their software. Various methods of describing software complexity are described in BS 5760 Part 8. Although many measures apply to code, a number can be used earlier in the life-cycle, for instance on specification and design documents. Many are supported by tools, although these have to be used with care. The most effective uses for static measures are:

a) To identify problem areas. For example, modules that show an anomalously high complexity compared to the typical values for a particular software system (especially when several measures are taken together) are likely to have a relatively higher fault density and require extra scrutiny;

b) To predict maintainability costs. Size and complexity measures, coupled with productivity data from the development life-cycle, can be used to estimate the effort needed to undertake maintenance tasks.

2.16 Subjective Requirements Evaluation

2.16.1 Evaluation of the software requirements is a type of internal review that may be carried out when preparing their proposal and as part of contract review and project risk assessment.

2.16.2 Subjective evaluation should be undertaken by the design team and the V&V team. Each group should evaluate each requirement on a scale from 1 to 5 as defined in **Table 1** (below).

(a) Designers' Evaluation	
1	You understand this requirement completely, you have designed from similar requirements in the past, and you should be able to develop a design from this requirement successfully.
2	There are elements of this requirement that are new to you, but they are not radically different from requirements that you have successfully designed from in the past.
3	There are elements of the requirement that are very different from requirements that you have designed from in the past, but you understand it and think you can develop a good design from it.
4	There are parts of the requirement that you do not understand, and you are not sure you can develop a good design.
5	You do not understand this requirement at all, and you cannot develop a design for it.
(b) V&V Team's Evaluation	
1	You understand this requirement completely, you have tested against similar requirements in the past, and you should be able to test the software against this requirement successfully.
2	There are elements of this requirement that are new to you, but they are not radically different from requirements that you have successfully tested against in the past.
3	There are elements of this requirement that are very different from requirements you have tested against in the past, but you understand it and think you can test against it.
4	There are parts of this requirement that you do not understand, and you are not sure that you can devise a test to address this requirement.
5	You do not understand this requirement at all, and you cannot develop a test to address it.

Table 1: Subjective Requirement Evaluation

2.16.3 The outcome of the subjective evaluation is a profile of the understanding and perceived novelty of the requirements. If the evaluation profile is predominantly '1s' and/or '2s', the evaluation is satisfactory and design work maybe allowed. If the profile is predominantly '4s' and/or '5s', the requirements should be clarified or the team composition adjusted, and the evaluation repeated before further work is permitted.

2.17 Testing

2.17.1 It is important to realise the limitations of testing as a method for achieving software reliability. In most software systems, there are so many combinations of inputs and internal states that it would be completely impractical to test them all. This means that at the point of acceptance into service, only a small proportion of the possible tests will have been carried out. More information on testability is contained in Def Stan 00-42 part 4[2].

2.17.2 Unit Testing and Module Testing. This is the process of testing individual software components and modules before they are integrated into the complete system. This level of testing is poor at predicting in-service software reliability since the tests may mirror mistakes that have been made in the programming. However, the results are available earlier in the development than those from the other types of test, and provide a general indication of project progress.

2.17.3 Structural Testing. Also known as "white box" or "glass box" testing, this uses the internal structure of the program to guide the testing. It may be possible to obtain estimates of fault density and reliability from structural testing. Structural testing strategies aim to cover a certain proportion of the "objects" that make up the software (this is known as the "test effectiveness ratio" or TER): for instance, the objects might be program statements, branches, "linear code sequence and jumps" or paths. Other things being equal, the higher the proportion of these objects that are covered, the lower the number of post-delivery faults is found to be.

2.17.4 Functional Testing. Also known as "black box" testing, this uses test data derived from the specification, and usually aims to exercise all the functions of the software. The test data is chosen to have a good likelihood of finding faults, for example by including values at the boundaries of the input domain.

2.17.5 Statistical Testing aims to determine the reliability of the system directly by testing it in a representative environment, and is discussed further in PtCCh52 paragraph 2.2.

2.17.6 Exhaustive Testing is in effect a type of analytical argument and is discussed in PtCCh52 paragraph 5.

2.17.7 Stress/Overload testing. Software failures frequently arise in overload conditions. When resources are limited, the software may be unable to complete an operation or it may perform too slowly to be useful in practice. Measurements on complex software systems show that failure rates can be orders of magnitude greater when resource utilisation is high. Stress/overload testing addresses such failures by concentrating on values that lead to high resource utilisation by operating the software with large values of data rate, message size, number of targets, number of users, etc.

2.18 Suitably Qualified & Experienced Personnel (SQEP) in Software Design

2.18.1 One of the most important factors in the achievement of reliable software is the quality of the development team. Attributes that should be considered include:

a) **Appropriate Skill Levels.** The team as a whole should possess formal training and/or project experience in all the techniques and methods to be applied. If special techniques (e.g. formal methods) are to be employed, an expert adviser should be available;

b) **Stability.** High staff turnover should be avoided, and most of the members, including the Team Leader, should have worked together on previous, successful projects;

c) **Attitude.** The team should have a "total quality management" (TQM) attitude to their work, and actively seek to avoid faults. They should have high morale and good motivation;

d) **Management.** The management and organisation should be demonstrably aware of human factors issues. Team members should have a planned career progression, adequate training and comfortable working conditions;

e) **Avoidance of "Groupthink".** A degree of independence should be included in software assurance activities to counteract the tendency of a closely-knit team to fail to notice shortcomings in its work.

2.19 Design Maturity

2.19.1 Software systems may exhibit reliability growth in the field, as reported failures lead to fault removal in subsequent releases, and software with an extensive operating history is likely to be more reliable than new software, provided changes have been well managed. In practice, the increase of reliability with design maturity is most marked in small systems and in those systems where the emphasis is on removing faults rather than increasing functionality. The reuse of such mature software designs is a way of achieving reliability provided that the reuse is possible without change and in a similar environment. Reuse can either be of the Contractor's own software, COTS software, or other pre-existing software.

2.19.2 A particular problem with reuse is that it has often been observed that an item of software can be very reliable when used in one context, but fail frequently when transplanted into another application. This is primarily because the mode of use may change and the resulting differences in the input data values can "hit" a software fault and cause an error. Figure 1(a) illustrates the case where the software does not fail despite a fault, because the input data does not invoke the faulty code; Figure 1(b) shows how a change in the input data causes the fault to become prominent. Thus it is necessary to show that the past operating experience is statistically "similar" to the current application so that similar levels of reliability can be anticipated. An alternative is to reuse software that has experienced a wide range of different modes of use that should give good coverage of the most typical parts of the input space, so that at least one prior mode of use must have involved the same input values, loadings, data rates, etc., as the current application.

2.19.3 In principle, software reuse does not need to be at the code level; specifications and designs can also be reused, with the advantage that they can be implemented on different hardware and using different programming languages and tools. However, there is little reuse at this level in practice, and no data on its effectiveness.

2.19.4 Reusable components should be provided with adequate documentation, including a precise specification. Formal specifications (see paragraph 2.11**)** will avoid many of the problems of trying to reuse ambiguously or incompletely defined software.

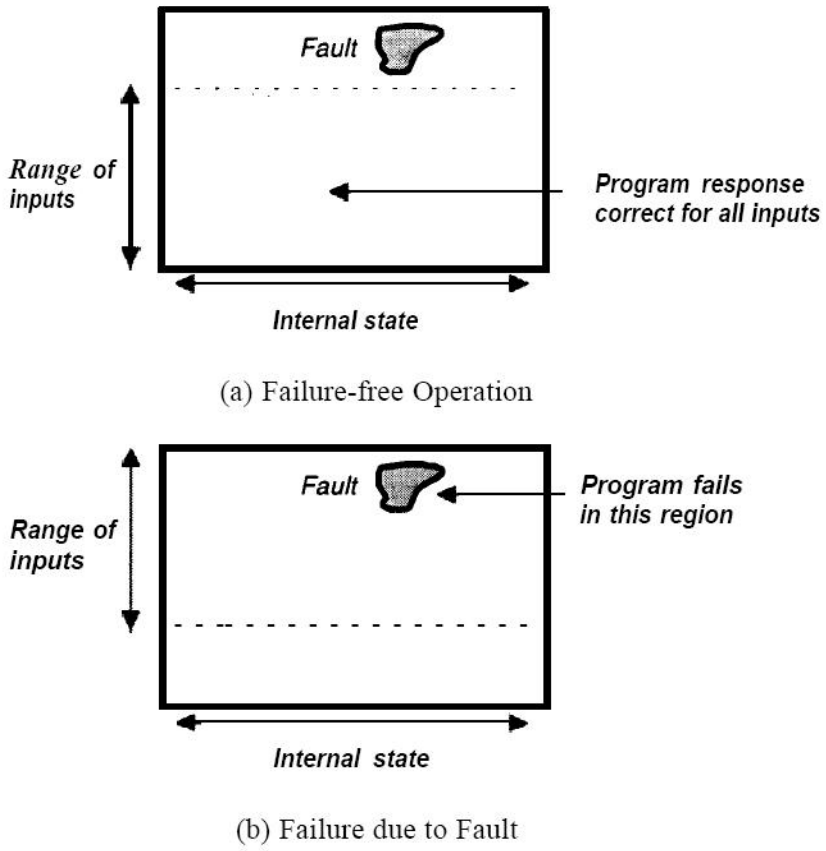

Figure 1: Apparent Size of a Software Fault

2.20 Analytical Arguments

2.20.1 Analytical arguments are essentially arguments that certain faults are *logically impossible.* In some circumstances, it may be easier to show by analysis that an event is logically impossible than to show by testing that it is very unlikely (e.g. has a probability of failure of 10^{-9} per hour). There is always a small doubt in analytical arguments because of assumptions about the environment, the soundness of the methods used to construct the arguments, etc.

2.20.2 Examples of analytical arguments are:

a) Analysis to show that the response time, throughput, capacity and resource usage areas specified. Note that tools may be available to assist this process;

b) Analysis of the effectiveness of fault detection (including BIT), fault tolerance and fail safety features, including analysis that the overheads from these features are within the specification;

c) The use of formal methods to show that certain properties are logically bound to hold of the software (examples of this might be that critical functions occur correctly, or that undesirable behaviour cannot occur);

d) The use of static analysis to show that coding errors have not occurred;

e) Reverse compilation of object code to show that compiler errors have not occurred;

f) Model checking to show that real-time failures (e.g. deadlock) cannot occur; and

g) Analysis to show that all stimuli have a valid response.

3 FAULT TOLERANCE

3.1 Where high reliability is required, it is likely to be necessary to design the system to detect and mitigate software failures. This is best done by appropriate design diversity at the system level (e.g. a programmable and non-programmable system to implement a critical function), but where this is impractical, a degree of fault tolerance can be added to the software itself by means of defensive programming and software diversity.

3.2 Defensive Programming

3.2.1 Defensive programming is the simplest form of fault tolerance. It works by placing checks in the software that the program variables are consistent and have not been corrupted (i.e. causing a divide by zero). If an inconsistency is detected, some error handling process is initiated.

3.2.2 Error Detection. Error detection may, for example, be simply by checks included in the program that the data is within a certain range, or may involve complex relationships between data items. The code to implement these checks may be generated automatically by the compiler from assertions included in the source code (these assertions can be derived from a formal specification), but more often the checks have to be coded by hand. Some languages, such as Ada, have exception handling features that transfer control to a handler program if a check fails. Another mechanism for error detection is to associate error detecting codes (e.g. checksums) with the data to detect external or internal data corruption.

3.2.3 A serious problem with defensive programming is that the checking code adds to the complexity of the software and slows it down, and therefore it should be introduced according to a carefully-defined strategy.

3.2.4 Error Handling. Error handling can be by backward error recovery, forward error recovery, fail soft or fail safe strategies. Error recovery is obviously preferable, but it can be hard to implement, especially if asynchronous, communicating programs are involved.

a) **Backward Error Recovery.** This is usually implemented by making copies (known as checkpoints) of the correct state at intervals, and restoring the state from the latest checkpoint if an error is detected. It is most useful against transient hardware failures and external faults;

b) **Forward Error Recovery.** This uses redundant information to repair corrupted data. Error detecting codes (i.e. Hamming codes) are often implemented with extra bits to enable error recovery. Database and file systems commonly use redundant pointers for error recovery. Diversity can also be used (see section 3.3);

c) **Fail Soft.** Here the design philosophy is not to mask failures but to reduce the performance of the system in a defined way. A fail soft philosophy may, for example,

be implemented by reducing the functionality, assuming that some partial operational capability can be identified, or by discarding data (e.g. alternate inputs);

d) **Fail Safe.** It maybe possible to handle errors by halting in a safe way. Examples of fail safety are seen in railway signaling, where all signals are set to danger if an error is detected in the interlocking system, and in nuclear power generation, where the reactor is shut down if the safe operating parameters are exceeded, or if an error is detected. Fail safety is not possible at the system level if high availability is required, but maybe possible at the subsystem level if it can be arranged for a faulty component to "fail silent" (i.e. cease producing output) and a backup subsystem to detect this and take over.

3.3 Software Diversity

3.3.1 Software diversity is a form of fault tolerance based on the use of two or more diverse programs to carry out critical functions, on the assumption that the same fault is unlikely to occur in all the versions. In its most elaborate form, diversity may involve three or four versions of the software, written by independent teams using different design methods and programming languages, and running on different processors.

3.3.2 Diversity enables failures of individual versions to be detected, and fault tolerance to be implemented, by using the output agreed on by the majority to control the system. If all the versions disagree, the system should fail soft or fail safe if possible.

3.3.3 Diversity is vulnerable to common mode faults affecting several of the versions, which limit the reliability gain and make it hard to predict. Research has shown that diverse implementations may suffer from common design errors, possibly because certain parts of the design are intrinsically difficult. It may also be difficult to maintain data consistency between the versions, especially if the software maintains an internal state.

3.3.4 Software diversity may be difficult to implement for both technical and managerial reasons. Shared activities make it very hard to maintain isolation between the teams, and it has been found that the costs of diversity increase rapidly with the number of versions. It is likely to be more cost-effective to concentrate resources on fault avoidance and detection during development, and diversity, if it is used, should be employed at a system, rather than software, level.

LEAFLET 51/0

BIBLIOGRAPHY

Standards

1. BS 5760 Part 8, "Reliability of systems, equipment and components. Guide to assessment of reliability of systems containing software", British Standards Institute, October 1998.
2. DEF STAN 00-42 part 4, "R&M Assurance Guide – Testability", Issue 1, MoD, 13th December 2002.
3. DEF STAN 00-56, "Safety Management Requirements for Defence Systems", Issue 4, MoD, 1st June 2007.
4. DEF STAN 00-60, Part 3, "Integrated Logistic Support, Guidance for Application of Software Support", Issue 3, MoD, 24th September 2004.
5. IEC 61508 (Part 3), "Functional Safety of Electrical/Electronic/programmable electronic safety-related systems". Part 3: software requirements.
6. The TickIT Guide, Issue 5, British Standards Institute, January 2001.

Papers

7. "What We Have Learned About Fighting Defects," Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS™02), IEEE Computer Society, Basili, Vic, Boehm, Barry, and others, 2002.
8. "Software Reliability Source Book," Data and Analysis Center for Software, Rome, DACS CD, NY, 2002.
9. "Results and Achievements from the DTI/EPSRC R&D Programme in Safety Critical Systems", Falla, Mike, Edited by Mike Falla, Motor Industry Software Reliability Association, November 1996.
10. "Planning Models for Software Reliability and Cost", Helander, M., Shao, M., and Ohlsson, N., IEEE Transactions on Software Engineering, Vol 24, Number 6, June 1998, pp 420-434.
11. "Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors", Herrmann, D., IEEE Computer Society, Los Alamitos, CA, 1999.
12. "Software Quality In 1997: What Works and What Doesn't", Jones, Capers, Software Productivity Research, 1997.
13. "System and Software Reliability Assurance Notebook", Lakey, Peter and Neufelder, Ann Marie, Rome Laboratory Report, Griffiss Air Force Base, Rome NY, 1997.
14. "Safeware: System Safety and Computers", Leveson, Nancy G., Addison Wesley Publishing Company, 1995.
15. "Software Reliability and Dependability: a Roadmap, in The Future of Software Engineering", Littlewood, B. and Strigini, L., State of the Art Reports given at the

22nd Int. Conf. on Software Engineering, Limerick, June 2000, (A. Finkelstein, Ed.), pp. 177-188, ACM Press, 2000.

16. “Handbook of Software Reliability Engineering”, Lyu, Michael, ISBN 0-07-039400-8, McGraw Hill, 1996.
17. "Operational Profiles in Software Reliability Engineering”, Musa, John D., IEEE Software, March 1993, pages 14-32.
18. “Software Reliability Engineering”, Musa, John D., McGraw-Hill Book Company, NY, 1999.
19. "Worldwide Reliability & Maintainability Standards”, PRIM-97, Reliability Analysis Center, IIT Research Institute / Reliability Analysis Center, Rome, NY, 1997.
20. "Reliability Modeling for Safety-Critical Software”, Schneidewind, N., IEEE Transactions on Reliability, Vol 46, Number 1, March 1997, pp 88-98.
21. “Software System Safety Handbook”, Joint Software System Safety Committee and EIA G-46 Committee, Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, U.S. Air Force, 1999.
22. "Programming Languages”, CrossTalk, Journal of Defense Software Engineering, Vol. 16 No. 2, February 2003.
23. “Practical Guide To Certification and Re-Certification of AAvA Software Elements Software For Programmable Logic Device”, issue 4.1, Ashley, July 2005
24. “Practical Guide To Certification and Re-Certification of AAvA Software Elements COTS Real-Time Operating Systems”, issue 4.1, Dobbing, July 2005