CHAPTER 15

SOFTWARE R&M ANALYSES

CONTENTS

1 INTRODUCTION

1.1 General

1.1.1 This Chapter discusses the techniques appropriate to the analysis, assessment and progressive assurance of the R&M characteristics of software. The aspects covered are:

- Techniques to be applied during each phase of the system's Life Cycle that assist the development and maintenance of software with adequate integrity;
- The application of techniques intended to assess the R&M characteristics of software.

1.1.2 Each of these two aspects will be discussed within the following sections, although the application of the techniques is often interactive and iterative.

1.1.3 It is not the intention of this chapter to address the techniques applicable to the development and maintenance of 'good quality' software. Such techniques fall within the scope of software engineering and the application of an appropriate quality system. The aim of the chapter is rather to identify where and how the software within a system may be integrated within the R&M programme of activities that may otherwise focus exclusively on the system hardware.

1.1.4 **Section 2**: provides a general discussion of what is meant by the term 'software' and the major attributes of and influences upon software and software reliability and maintainability. Section 2 is supported by Tables 1 and 2 which summarise the difference between software and hardware in terms of reliability and maintainability respectively.

1.1.5 **Section 3**: provides a cross reference from this chapter to other sections of the R&M manual which discuss R&M techniques. Section 3 is supported by Table 3 which provides a brief summary of the applicability of each R&M technique to software. More detailed guidance is available in Def Stan 00-42, Part 2 (Ref 1).

1.1.6 **Section 4**: provides an outline of the weaknesses of current reliability prediction techniques applied to software and provides a discussion of how weaknesses may be overcome. Section 4 is supported by Table 4 which contains the basic details of four reliability prediction models.

1.1.7 **Section 5**: provides an example of one method of software reliability prediction considered to represent best practice and currently employed in industry.

2 ABBREVIATIONS

BITE(E)	Built in Test Equipment (Effectiveness)
CADMID	Concept, Assessment, Demonstration, Manufacture, In Service, Disposal
COTS	Commercial Off The Shelf
CSCI	Computer Software Configuration Item

Def Stan	Defence Standard
DRACAS	Data Reporting Analysis & Corrective Action System
ESM	Electronic Surveillance Measures
FME(C)A	Failure Modes, Effects (& Criticality) Analysis
HCI	Human/Computer Interface
IEC	International Electrotechnical Commission
KSLOC	Thousand Lines Of Source Code
LED	Latent Error Density
MMI	Man/Machine Interface
MTBF	Mean Time Between Failure
R&M	Reliability and Maintainability
SFD	Software Fault Density

3 SOFTWARE

3.1 Introduction

3.1.1 Software performs many different functions within a system and this section provides a discussion of the major attributes of software and influences upon software development, and hence R&M attributes of software. The attributes of software within a system are different to those of hardware and the factors behind hardware and software R&M are also diverse. Table 1 provides a brief comparison between software and hardware reliability and Table 2 provides a brief comparison between software and hardware maintainability. The following high level headings identify major aspects associated with software:

- Life cycle;
- Timing requirements;
- Safety issues;
- Functional type;
- Development or procurement policy;
- Software development processes and languages.

3.1.2 Each of these topics is discussed below.

3.2 The Life Cycle of Software

3.2.1 Since software in its various forms is now integral to all aspects of the life cycle management of a system, it requires development and management procedures that do not merely mirror those of associated system hardware.

3.2.2 System life cycles are now defined by MoD under the CADMID cycle:

- **C**oncept;

- **A**ssessment;
- **D**emonstration;
- **M**anufacture;
- **I**n-Service;
- **D**isposal / Update.

3.2.3 Part C of this Manual identifies many techniques that are available to manage, design and assess equipment R&M to the desired level through various stages of the CADMID cycle. These are discussed in Table 3 at the end of this Chapter.

3.2.4 Characteristic software types are discussed below in the context of a generic system programme. Clearly, not all types of software will be required in every system. Furthermore, the point at which software development and integration into the system takes place (assessment, demonstration and manufacture phases) will vary according to the programme needs. However, a common thread that can be identified is that all software will require some level of support during the operational life of the system.

3.3 Temporal Aspects

3.3.1 The term “Temporal” relates to the time constraints placed on the function provided by the software. Three categories are proposed:

- Real time;
- Constrained, near real time;
- Information Management Tools.

3.3.2 **Real Time** This type of software is required to provide responses to inputs almost instantaneously. Input data is processed as it is received and provides the desired output function against a well-defined and constrained requirement.

3.3.3 Examples of real time software functions are sensor signal processing e.g. a radar receiver, where all the received data must be conditioned and beamformed on a cyclic and continual basis.

3.3.4 **Constrained Near Real Time** These types of software functions have relaxed requirements in comparison to real time processes, but must still act and deliver their functionality within pre-defined or user perceived limits of acceptability. The time taken to provide an output may be a function of the complexity or quantity of the data presented for processing.

3.3.5 Examples of this category are:

- data fusion where more time is required to analyse the data as the quantity of valid data sources increases;
- narrow band sonar data processing;
- correlation of radar signals for track identification;

- messaging systems where the time taken to deliver a message will increase with the size of the message being sent and the level of activity on the system.

3.3.6 Such systems will have calculated or probabilistic requirements, e.g. an increased time is permitted for the production of processed data in proportion to the number of input data sources or it is stated that 50% of messages shall be delivered in a particular time, 90% in another time and all messages within an upper time limit.

3.3.7 Information Management Tools This category addresses the common everyday software tools used in an electronic environment and covers such tools as word processors, spreadsheets, databases and general management packages.

3.3.8 There are usually no fixed timing requirements placed on Information Management Tool software other than the users perceived level of acceptable performance since the environment within which the software operates is overly dependant upon each installation and user’s operating method. A faster processor or additional memory will usually increase the ‘speed’ of the application and operating a system with several open applications will impose timing penalties.

3.4 Application in Safety Related Systems

3.4.1 General Software may be considered as:

- Safety related; and/or
- Operationally critical; or
- Non-critical.

3.4.2 Software integrity for safety related software must be higher than for software with no safety implications. Similarly, as the consequence of failure increases then so do the requirements upon the software developer. Software safety assurance is process based and does not concern the direct measurement of reliability, but instead defines the necessary development methods and tools that should be used in the development of safety related software. It should be noted that if the failure rate of the safety related software can be demonstrated in service then the underlying Latent Error Density (LED) is almost certainly too high! The LED of an item of software is an expression of the number of coding and logic errors contained within the item. LED does not provide a measure of how significant these errors may be to the function of the software or criticality of a resulting software failure during operation.

3.4.3 Safety Related Software Safety related software is code that either contributes to the maintenance of a safe condition or state or whose mis-operation could result in an un-safe event.

3.4.4 The first of these sub-categories relates to software that either controls or monitors the operation of a system or initiates action in the event that an unsafe condition arises. Examples of this are software functions that monitor or regulate the state of a chemical reaction, warns of an over temperature condition in an engine or move the control surfaces of an aircraft.

3.4.5 The second type is software that controls an inherently hazardous function where an un-demanded operation is undesirable. The untimely release of a weapon or movement of a piece of machinery falls into this category.

3.4.6 Operationally Critical This category addresses software that may or may not be safety related and whose failure to operate could increase the risk of a hazardous situation developing into an undesired event. There are many examples of this type of software e.g. a decoy system where the failure to operate in certain circumstances may not be safety critical, but the software reduces the probability of preventing damage to a platform. Similarly, the ESM system that programmes the decoy is operationally critical if the decoy is to achieve maximal effectiveness and may, in times of war, be safety related.

3.4.7 Non-critical This category encompasses software that has no safety or operational related characteristics and may include such applications as data logging or management information systems and tools.

3.4.8 Safety assurance techniques to minimise the Latent Error Density of software may also be applied to improve the reliability of non safety related software. However, these measures are essentially quality assurance techniques and are not covered in this manual. They are defined in detail in the “Functional safety of electrical/electronic/programmable electronic safety related systems” International Electrotechnical Commission (IEC) standard IEC 61508 (Ref 2) and the Defence Standard 00-55 “Requirements for Safety Related Software in Defence Systems” (Ref 3).

3.5 Functional Type

3.5.1 General This major heading addresses the type of function performed by the software. Each category identified has particular characteristics which may generate a level of inherent coding error. However, no specific values are given as this is an area that cannot be agreed by leading authorities in the software industry. The sub headings proposed are as follows:

- Algorithmic;
- Automatic processing;
- MMI/HCI.

3.5.2 Algorithmic This type of software performs well-defined functions, usually within well-defined time constraints and on a regular, cyclic basis. Examples of this type of module are sensor front-end signal processing or beamforming functions or cryptographic processes.

3.5.3 An Algorithmic process accepts pre-defined data formats, performs a specific numeric combinational function and outputs the processed data for a preceding higher order process.

3.5.4 As a result of the numeric basis of the functions performed and the well defined processed required, algorithmic functions tend to be compact code that can be developed with a low probability of error, and a low resulting LED.

3.5.5 Automatic Processing Automatic processing systems perform functions on data in accordance with a set of defined rules or instructions. They are of a higher level of processing to algorithmic functions since the timing, quality and quantity of input data sources and the processing applied to the received data may vary.

3.5.6 Examples of automatic processing are many and varied e.g.:

- Track extraction from sensor signals;
- Sensor data fusion;
- Computer aided detection and classification;
- Message routing;
- Engine management systems.

3.5.7 The wide range of functions required of automatic processors results in software that mat be less robust than algorithmic processes and hence it is more prone to contain errors.

3.5.8 **MMI** The Man Machine Interface or Human Computer Interface (HCI) software encompasses the means by which operators interact with a system, configuring the functions required and interrogate the processed information.

3.5.9 Many MMI devices have a variety of input and output facilities:

- Keyboard;
- Pointing device – mouse, tracker ball, light pen;
- Voice recognition;
- Text outputs;
- Graphical outputs;
- Audio – warnings, prompts, multimedia;
- Status lights and enunciators.

3.5.10 The wide ranging input sources, both from the operator and processed data to be supplied to the operator result in the timing and processing requirements of the MMI software being difficult to predict and manage. There are often rapidly changing priorities in respect of the order that information must be provided to the operator and the sequence and complexity of inputs from the operator may be difficult to predict and manage. These characteristics often render the MMI software more error prone than any other part of any system.

3.6 Development Type

3.6.1 **General** This category addresses the development source of the software and often presents another parameter suitable for Trade-off Studies. Although only two distinct categories are cited, many systems incorporate a combination of both types of software:

- Bespoke;
- Commercial Off The Shelf (COTS).

3.6.2 **Bespoke** Bespoke software is code procured and written to satisfy a specific requirement. By its nature, it tends to be “special to type” and “one-off”.

3.6.3 There are circumstances that dictate the need for bespoke software e.g. front-end signal processing for a sensor where there may be new functional requirements or particular timing or scaling requirements.

3.6.4 Bespoke software is generally less robust and more expensive than COTS products of a similar nature and level of functionality due to their low volume, specialist nature.

3.6.5 COTS COTS products are those that are commercially available and are generally proven, tested and complete at the time of purchase.

3.7 Software Development Processes and Languages

3.7.1 General There is a proliferation of software languages that may be employed to develop the executable code deployed in a system. In many cases the choice of language is determined by the application or function required. However, this is not the rule.

3.7.2 Three types of sub-categories are employed to define the development process and resulting classification of the software code:

- High Integrity Software;
- High Level Languages;
- Low Level Languages.

3.7.3 Each category is appropriate to satisfy different types of requirement and each will impose different development environment and management requirements. There are high integrity programming languages but these can only be used effectively for safety critical applications with the appropriate development life cycle. However, low level languages can also be used in high integrity applications e.g. for special device drivers.

3.7.4 Many studies have been conducted to demonstrate the impact of language upon software reliability. The results of these studies have not been universally accepted but certain general rules and guidelines have been established. One set of guidelines is contained in IEC 61508 which states the suitability of languages for different levels of safety integrity. It states that highly structured languages such as ADA or Modula-2 are preferred for high integrity software applications. Def Stan 00-42 Part 3 provides guidance on progressive assurance pertaining to software and the associated R&M Case.

3.7.5 High Integrity Software High Integrity software is required to support safety related and, by choice, operationally critical functions. The development environment for this type of software supports the capture and detailed documentation of the development process. This embraces

- A formal method for the decomposition and capture of requirements;
- Structured coding methods;
- Detailed rules for memory usage and management and variable definition and initialisation;
- Strict peer review processes and code walk-throughs;
- The use of static and dynamic testing.

3.7.6 The development risks of High Integrity code can be reduced by selection of the language employed e.g. the ADA language incorporates coding structures and mechanisms that assist and guide in the development of this type of software.

3.7.7 High Level Languages The majority of software is developed using these types of languages. The selection of the particular language to use may be dictated by the required application e.g. the development of a MS Windows program may be best achieved by the use of a 'visual' language, Visual Basic, Visual C or C++, etc. The use of techniques such as Object Oriented code or Rapid Application Development makes the development easier to manage and control, and is generally more cost effective.

3.7.8 Particular applications may suggest the use of specialised languages e.g. the use of Prolog or List for artificial intelligence applications whilst heavily computational applications may suggest the use of FORTRAN.

3.7.9 Low Level Languages Low Level languages are at the opposite end of the spectrum to the High Integrity software languages. Low Level languages encompass assembly or machine code routines.

3.7.10 The use of these types of language is usually dictated by the following considerations:

- the need for speed such as the sensor signal front-end processing;
- where space for the storage of the software is constrained;
- a particular function is not available in the High Level language being used for the majority of the software development programme.

4 SOFTWARE APPLICATION R&M TECHNIQUES

4.1 Introduction

4.1.1 The aim of this section is to provide guidance with regard to the techniques available to the R&M practitioner wishing to integrate an evaluation of the software aspects of a system into an overall system assessment.

4.1.2 Many acquisition programmes fail to address the R&M aspects of software adequately with unfortunate through life consequences. The proportion of a system's functionality realised in software has risen dramatically over the last decade and the effect of software failures on a system's R&M characteristics has increased commensurately. Indeed, for many complex systems, software failures constitute by far the largest contributor to system unavailability, sometimes exceeding 80% of all reported in-service incidents.

4.1.3 Software should be addressed within system R&M analyses. These analysis techniques are discussed in other sections of the R&M Manual:

a) High Integrity Specification - PtCCh6;

b) High Integrity Design - PtCCh10;

c) Redundancy Optimisation - PtCCh11;

d) R&M Checklists - PtCCh23;

e) Fault Tolerance Analysis - PtCCh27;

f) Fault / Success Tree Analysis - PtCCh29;

g) Allocation / Apportionment - PtCCh5;

h) Reliability Block Diagrams - PtCCh30;

i) Availability Prediction - PtCCh35;

j) Reliability Prediction - PtCCh36;

k) Maintainability Prediction - PtCCh37;

l) Failure Mode, Effects and Criticality Analysis - PtCCh33;

m) R&M Design Criteria - PtCCh12;

n) R&M Plans & Programmes - PtCCh49;

o) Trade-Off Studies – PtCCh4;

p) Part Derating - PtCCh7;

q) Critical Item List - PtCCh9;

r) Test, Analyse and Fix - PtCCh13;

s) Step Stress Testing - PtCCh14;

t) Reliability Growth Testing - PtCCh15;

u) Tolerance Analysis - PtCCh20;

v) Built-In-Test Effectiveness Analysis - PtCCh21;

w) Dependent Failure Analysis - PtCCh28;

x) Human Reliability Assessment - PtCCh32;

y) R&M Aspects of Sub-contracts - PtCCh48;

z) Data Reporting, Analysis and Corrective Action - PtCCh18;

aa) R&M Demonstration - PtCCh39 - 41;

bb) In-Service Data Collection - PtCCh47;

4.1.4 The relevance of software analysis in the above techniques is shown in Table 3. While it can be useful to apply the above techniques to software within a system procurement it should be noted that traditional approaches to software R&M have generally been unsuccessful. Until quite recently there has been little evidence to suggest that a pseudo random software failure rate may be assumed. This is in part because:

- There have been few efforts to systematically measure software reliability and this has led to software development promising future enhancements to improve the situation, such as structured designs and programming in the 1970s, CASE in the 1980s and Object Oriented methods in the 1990s. If there have been improvements through these approaches the number of software code errors may have reduced but complexity of the code has risen to cancel out any overall reliability improvement.
- There is an apparent difficulty in translating a Latent Error Density residing in each piece of software into a rate of occurrence of failures when the software comes to be used in a real time environment.

4.1.5 The rate of occurrence of failures is a function of the Latent Error Density and the usage characteristics of the software such as variability of duty cycles and capacity required. The major impact of the operational environment or domain upon in service reliability of software will always confound any efforts to express software reliability in analytical terms that consider the software only. For this reason a few organisations have attempted to address software reliability holistically by considering the software characteristics within the proposed operational and organisational context. An outline of such an approach is provided in Section 4.

5 MODERN SOFTWARE RELIABILITY ASSURANCE TECHNIQUES

5.1 Introduction

5.1.1 The field of software reliability prediction is still in its infancy, with a multiplicity of modelling approaches available. A few such models are discussed briefly in Table 4. Each model operates from a different set of premises and assumptions, e.g. distribution of error detection in time, input or derived parameters, software characteristics, development environment. There is a wide range of mathematical and statistical complexity within the models and a similarly wide range of data processing complexity required to implement the models and derive the characteristic parameters. Because of this, it is worth noting that when an assessment is made to select a model to 'fit' a set of data it will not always follow that the use of a complex mathematical relationship will be required. Sometimes the accessibility and ease of interpretation associated with a less statistically esoteric model is beneficial given that the ultimate aim of the activity is for System assessment.

5.1.2 The terminology and associated definitions used here are in accordance with those in BS5760 (Ref. 5) and are summarised below

- **Software Fault:** design fault in a software component;
- **Software Failure:** system failure due to the activation of a software fault in a software component. (Note: failure is defined as the item ceasing to perform a required function or provide a required service, in whole or in part.);
- **Activation(of a fault):** the combination of circumstances in which a software fault gives rise to a software failure.

5.1.3 The need for Software reliability estimation as part of the R&M and Safety activities should be embodied into the Project from inception, with the Prime Contractor being required to present their methodology for all types of software and for all Sub-Contractors.

5.1.4 The majority of Software Reliability models currently available attempt to predict the reliability in the later stages of the lifecycle (Integration Test and beyond) with relatively few applicable to the earlier (Design and Development) phases (Ref 4). Unfortunately, there are no ‘off the shelf ‘solutions, enabling predictions on the basis of generic models for these early stages. Software engineering is still at the stage where each organisation’s processes are unique to that organisation, and even within the organisation, relationships observed on one set of projects will be applicable only to other ‘similar’ projects. Thus, the use of fault data methods is dependent on the availability of data from previous projects (Ref 5).

5.1.5 In most instances the software lifecycle for reliability prediction purposes spans both the early and late stages of the life cycle i.e. Design & Development, Integration & Test and Operational Service. The software can thus be considered to be operated in two different environments - Development & Test and Operational. The data requiring gathered in each environment is different and the transition point between them is completion of acceptance into service. Due to this discontinuity, it is accepted that a single model, directly predicting reliability in one environment from data gathered in another, cannot be applied. Therefore a two model approach may be adopted with the results of one being fed into the other as follows:

- For the early part of the life cycle (pre Acceptance, Design and Test) a Phase based Software Fault Density (SFD) model is used to predict the Latent Error Density (LED) remaining in the Software at Acceptance;
- For the latter part of the life cycle (Acceptance into service, Operational) a Time based Software MTBF Prediction model maybe used to predict the future MTBF of the Software, using the LED at Acceptance, obtained from the Software Fault Density Model, as the initial input/start point.

5.1.6 Section 5 provides an example of how the two model approach may be adopted. It should be noted that the example given in Section 5 is not the only method of software reliability prediction, but is an example of current ‘best practice’.

6 SOFTWARE RELIABILITY PREDICTION BEST PRACTICE

6.1 General

6.1.1 This section provides details of how the two model approach has been adopted in industry as a reliability assurance technique for complex software. Section 5.2 discusses the application of the SFD within the early part of the software life cycle. Section 5.3 shows how the fault discovery profile generated by first model is used to predict software reliability.

6.2 Software Fault Density Model (SFD)

6.2.1 John Musa of AT&T Bell laboratories (Ref 8) demonstrated the basic assumption, underlying the use of the Software Fault Density Model, that the number of software faults revealed is a function of execution time and the total number of faults at the start of testing. By assuming that the execution time is a linear function of the testing period(s) Trachtenberg found that Musa's equation is essentially a Rayleigh curve. This hypothesis is supported by empirical historical evidence gathered from software development projects showing that if more faults are revealed and removed in the earlier development phases, fewer will remain in later stages which in turn yields better quality i.e. code with a lower LED value. (Refs 4,6)

6.2.2 Gaffney and Davis of the Software Productivity Consortium developed this concept into a Phase Based model (Ref 7) which uses the fault statistics generated during the phases of technical review of requirements, design, integration and test to predict the LED value remaining in the code at entry into Operational service. It is this model which has been adopted as the basis of the Software Fault Density Model (SFD) in industry.

A summary of the underlying assumptions of the model is as follows:

- The development effort's staffing level and expertise is directly related to the number of faults discovered during the development phase;
- Common/equivalent development standards, quality and rigour is applied by the Software development teams;
- The fault discovery curve is monomodal (i.e. it has no more than one turning point);
- Code size estimates are available during the early phases of a development effort. The model expects that fault densities will be expressed in terms of number of faults per thousand lines of source code (KSLOC), which means that faults found during the requirements analysis and software design will have to be dimensioned by the code size estimates;
- The fault metrics applied are for those faults that would result in failure of the software Computer Software Configuration Item (CSCI) i.e. those that require a change in the source code to correct. This results in the LED prediction obtained being for those faults that would result in failure of the CSCI.

The fault discovery profile of the model can then be represented by the following discrete function

$$V_t = E[\exp(-(t-1)^2/(2a^2))-\exp(-t^2/(2a^2))]$$

where

V_t : Estimated No. of discovered faults per KSLOC during Phase t

t : Phase Index No:

1. High level Design Inspection;
2. Low Level Design Inspection;
3. Code Inspection;
4. Unit Test;
5. Integration Test;
6. System Test.

E : Expected Total Programme Lifetime fault content expressed as Faults/KSLOC

a : Fault Discovery Phase constant, being the peak of a continuous curve fit to the failure data. This is the point at which 39% of the faults have been discovered.

6.3 Software Reliability Prediction Model

6.3.1 A basic execution time model for software reliability was developed by John Musa based on an exponential failure intensity function (Ref 8). Musa recommended the use of this model if

- it is required to predict reliability before program execution is initiated and failure data observed;
- the program is changing over time as the failure data is observed.

6.3.2 However, individual units of software may consist of different sections/classes; the failure rates of each, while still being exponential in form, can vary according to their different natures. Software natures are discussed earlier in section 2. To account for this, Ohba and others developed Musa's basic 'classical' model into the Hyperexponential Model (Ref 9). If only two classes of software are identified, the model is known as the modified exponential software reliability growth model (Ref 10).

6.3.3 Typical assumptions required for these models are as follows:

- The times between failures are piecewise exponentially distributed i.e. the system failure rate for a single fault is constant (hence the model belongs to the Exponential class of reliability models)
- The quantities of resources (e.g. number of fault identification/correction personnel and computer time) that are available are constant over a segment for which the software is observed
- Fault identification personnel can be fully utilised and computer utilisation is constant
- For each class of software,
    - ➢ the rate of fault detection is proportional to the current fault content and complexity of the software
    - ➢ the fault detection rate remains constant over the intervals between fault occurrence (i.e. the external influences which cause a fault to be revealed occur randomly)
    - ➢ a fault is corrected instantaneously without introducing new faults into the software (i.e. quick fixes and/or workarounds are applied immediately, with permanent fixes being applied as part of a Build update)
- Software changes (CRs and Permanent Fault fixes requiring code updates) occur as new Builds of the software which are implemented at regular intervals. The code updates of these builds add faults to the 'baseline' code which follow their own independent fault discovery curve

6.3.4 The fault discovery curve for one hyper exponential model for each software class of a CSCI can therefore be expressed as:

$$N_t = N_o[\exp(-kUt)]$$

where

N_t : Number of faults discovered by time t

N_o : Initial number of Latent Errors as obtained from the SFD model

k : Decay parameter, obtained from developers previous experience.

U : Usage Factor (based on number of instantiations of the code running concurrently in System)

t : Elapsed Time

6.3.5 The LED value obtained from the SFD model (N_o) is only for those faults which would cause the software to fail. Hence the fault discovery prediction (N_t) is similarly only those that cause failure. Hence, the Mean Time Between Failure (MTBF) curve for each class of the CSCI software can then be predicted by applying the standard calculation for observed MTBF i.e. dividing the Equipment Operating hours accumulated during the time interval by the number of failures discovered in the interval as follows

$$MTBF_t = (E.\delta t)/(N_{(t+\delta t)} - Nt)$$

where

$MTBF_{t\,=}$ MTBF at time t

δt = Time interval being observed (in hours)

N_t = Number of failures discovered at time t

$N_{(t+\delta t)}$ = Number of faults discovered at time (t+δt)

E = Number of instantiations of the software operating in the time interval

6.3.6 The MTBF of the CSCI as a whole is then calculated by combining the MTBFs of its constituent classes in accordance with the normal mathematics of Series System Reliability calculation as follows

$$MTBF_{tCSCI} = 1/\sum_{i=1}^{i=2}(1/\,MTBF i)$$

where

$MTBF_{tCSCI}$ = MTBF of the whole CSCI at time t

$MTBF_{ti}$ = MTBF of the i’th software class of the CSCI at time t

6.3.7 Inclusion of Build updates of the CSCI is accomplished by applying the a similar process of MTBF calculation and subsequent inclusion of that MTBF into the CSCI MTBF as described below.

6.3.8 At each Build update, the fault discovery curve for the updated (added) code is calculated using the equation of para 4.1.4, and the value of N_o obtained from the SFD model for the updated code.

6.3.9 The MTBF curve for the Updated portion of the code alone is then calculated using the equation of para 4.1.5.

The MTBF of the whole, updated, CSCI is then calculated by adding the Updated code MTBF to that of the existing CSCI MTBF (as per the normal mathematics for Series Reliability calculation), with the adjustment that the origin of the Update code MTBF curve is offset to the Update Date. This results in the following equation :

$$\mathrm{MTBF}_{\mathrm{tCSCIU}} = 1 \Big/ \sum (1/\mathrm{MTBF}_{\mathrm{tCSCI}}) + (1/\mathrm{MTBF}*_{tU})$$

where

$\mathrm{MTBF}_{\mathrm{tCSCIU}}$	=	MTBF of the whole, updated, CSCI at time t
$\mathrm{MTBF}_{\mathrm{tCSCI}}$	=	MTBF of the whole (exc. update) CSCI at time t
$\mathrm{MTBF}^{*}_{\mathrm{tU}}$	=	MTBF of the CSCI Update code at time (t-b)
T	=	total time since acceptance
b	=	Code Update time post SAT

6.3.10 This process is repeated for each Build Update, thereby summing the MTBF curves of the Initial CSCI release with those of each Update release (offset to the release dates) to generate the total CSCI MTBF prediction.

Hardware	**Software**
1. Failures can be caused by deficiencies in design, production, use and maintenance.	1. Failures are primarily due to design faults, with production (copying), use and maintenance (excluding corrections) having negligible effect.
2. Failures can be due to wear, or other energy-related phenomena. Sometimes a warning is available before failure occurs.	2. There is no wearout phenomenon. Software failures occur without warning. However, software may exhibit “pseudo wear out” at system level if the system is not adequately maintained. Failure rates may rise with time due to degraded quality of inputs, increased complexity due to software patches and unremoved files. This attribute is particularly true for complex data handling software applications.
3. Repairs can be made which might make the equipment more reliable.	3. Failures cannot be anticipated and there is no such repair. The only solution is redesign (reprogramming), which, if it removes the fault and introduces no others, will result in higher reliability.
4. Reliability can depend upon burn-in or wearout phenomena, ie. failure rates can be decreasing, constant or increasing with respect to operating time.	4. Reliability improvement with time may be affected, but this is not an operational time relationship, it is a function of the effort put into detecting and correcting faults.
5. Reliability is mainly time-related, with failures occurring as a function of operating (or storage) time.	5. Reliability is dependent upon time or activity based. Failures due to activity occur when a certain program step or sequence of steps is executed.
6. Theoretical predictions of reliability can be made from knowledge of design and usage factors.	6. Theoretical predictions of reliability can be made from knowledge of design, usage and organisation factors. This issue is discussed in Section 4.
7. Failures can occur to components of a system in a pattern that is to some extent predictable from the stresses on the components, and other factors.	7. Failures are not usually predictable from an analysis of each statement. Faults are likely to exist randomly throughout the program and any statement may be in error.

Hardware	Software
8. Reliability is related to environmental factors.	8. The external physical environment does not directly affect Reliability. However, dynamic inputs may vary due to environment e.g. asynchronous reporting into a complex system may be dependent upon environmental factors where the weather conditions may affect the range and scale of data inputs from other different platform types. The change in input timing and resulting software path changes may affect the reliability of the software.
9. Reliability can be sometimes improved by redundancy.	9. Reliability cannot be improved by redundancy if the parallel program paths are identical, since if one path fails the others will have the same fault. It is possible to provide redundancy by having parallel paths, each with different programs written and checked by different teams.

Table 1 Comparison of Hardware and Software Reliability

Hardware	Software
1. Various depths of maintenance depending upon system and skill levels available to user. Basic faults may be rectified at 1st line with simple change outs at 1st or 2nd line.	1. Little 1st line maintenance other than system resets or reboots. These take little time. System resets or reboots are not always necessary. It may be possible to download new versions of failing modules while the system remains 'live' – common in industrial control systems. Only other maintenance will be 4th line code re write or "patches" which typically may take days, weeks or longer to effect.
2. 1st and 2nd line maintenance tasks may take hours and represent a significant contribution to lost availability.	2. 1st line resets and reboots generally take little time. Availability more likely to be impacted by poor reliability rather than reset time.
3. May require sophisticated tooling, jigs	3. Diagnostic program/monitoring software

Hardware	Software
and/or lifting equipment at all maintenance lines to effect repair.	may be used.
4. Commissioning may often be effected by simple function checks.	4. Commissioning may require complex system simulation.

Table 2 Comparison of Hardware and Software Maintainability

R+M Analysis Technique	Cross Reference	Software Applicability
High Integrity Specification	Part C, Chapter 6;	Specification of R&M requirements for software may be necessary if R&M performance is to be specified below system level into functional block level or if significant software content suggests that software reliability may dominate the system reliability. The same considerations should be made for software requirements as for other systems requirements, i.e. consider the operating environment, criticality of software failure, requirements for diagnostic and test coverage
High Integrity Design	Part C, Chapter 10;	If high integrity design is required for the software then the most useful guidance is provided by the design standards for Safety Related software, IEC 61508 and Def Stan 00-55.
Redundancy Optimisation	Part C, Chapter 11;	Redundancy within software can be illusory if similar systems and code are used on redundant pathways. However, redundancy optimisation should consider the impact of software. This will typically be aided by assessment techniques such as reliability modelling and particularly dependent failure analysis (see below).

R+M Analysis Technique	**Cross Reference**	**Software Applicability**
R&M Checklists	Part C, Chapter 23;	While R&M checklists are used infrequently, when employed, software should be addressed if it forms a significant part of the system and is likely to be a threat to system capability. Particular areas that may be addressed would be the maintenance of the software covering all aspects from re boot to redesign
Fault Tolerance Analysis	Part C, Chapter 27;	Wherever possible fault tolerant software should be produced. Capability can be assessed by the review of design features such as the capability of the software to handle invalid inputs.
Fault / Success Tree Analysis (FTA/STA)	Part C, Chapter 29;	Generally software should be included as basic events in FTAs where the software provides a significant part of the system functionality.
Allocation / Apportionment	Part C, Chapter 5;	If an allocation is to be produced then generally software should be included if a realistic estimate can be obtained of the R&M characteristics at an early stage of the project. In many instances, it may not be practical as this activity should be conducted very early in the assessment phase when little numerical information will be available for the software. Particularly true for lengthy projects where supporting hardware may change before the contract is let.
Reliability Block Diagrams (RBDs)	Part C, Chapter 30;	Generally software should be included as specific functional blocks in RBDs where the software provides a significant part of the system functionality. It may not be necessary to explicitly model software in higher (system) level models.
Availability Prediction	Part C, Chapter 35;	Generally software should be included as specific functional blocks in availability predictions where the software provides a significant part of the system functionality. It is unlikely to impact the availability due to short reset times, but these issues should be recognised and included.

R+M Analysis Technique	**Cross Reference**	**Software Applicability**
Reliability Prediction	Part C, Chapter 36;	Generally software should be included as specific functional blocks in availability predictions where the software provides a significant part of the system functionality. On complex systems with considerable functionality provided by software (such as communications and guidance systems) it is likely that the reliability will be dominated by software failures.
Maintainability Prediction	Part C, Chapter 37;	Generally maintainability prediction will be aimed at minimising the hardware repair and servicing downtime by the provision of access and tooling. There is little to be gained from including software in such analyses. However, where the maintainability prediction is used to build up an availability prediction then the full system (including software) should be considered.
Failure Mode, Effects and Criticality Analysis (FMECA)	Part C, Chapter 33;	Generally software should be included as specific elements in FMEA and FMECA where the software provides a significant part of the system functionality.
R&M Design Criteria	Part C, Chapter 12;	Generally, R&M design criteria should include key aspects of software design that may impact system R&M performance. Design criteria may address software for items such as; existing (commercial?) code or fault tolerancing techniques
R&M Plans & Programmes	Part C, Chapter 49;	Software should be included in an R&M plan to cover areas such as software fault tolerance, software testing, reliability growth or monitoring.
Trade-Off Studies	Part C, Chapter 4;	The impact of software component of systems should be included in trade off studies, this may be between two alternative software solutions or between two systems with vastly different software composition.
Part Derating	Part C, Chapter 7;	This technique is not applied to software.

R+M Analysis Technique	**Cross Reference**	**Software Applicability**
Critical Item List	Part C, Chapter 9;	Generally software should be considered for inclusion on the Critical Items List where the software provides a significant part of the system functionality. Reasons for inclusion of the software on the CIL may be complex or novel design or the fact it provides.
Test, Analyse and Fix	Part C, Chapter 13;	This technique may be applied to good affect on software in complex systems. The modern reliability prediction and assessment techniques are derivations of Test Analyse Fix or reliability growth principles where initial estimates are gradually validated and errors removed to improve the software reliability.
Step Stress Testing	Part C, Chapter 14;	This technique is not applied to software.
Reliability Growth Testing	Part C, Chapter 15;	This technique may be applied to good affect on software in complex systems. The modern reliability prediction and assessment techniques are derivations of Test Analyse Fix or reliability growth principles where initial estimates are gradually validated and errors removed to improve the software reliability.
Tolerance Analysis	Part C, Chapter 20;	This technique is not applied to software.
Built-In-Test Effectiveness (BITE) Analysis	Part C, Chapter 21;	This technique is not directly applicable to software. However, in most cases the provision of BITE in a system will be enabled by software and status monitoring or sensors.
Dependent Failure Analysis	Part C, Chapter 28;	Software is a major cause of dependant failures. Multiple channels relying upon the same software provide little or no redundancy. It is therefore important to include software within dependant failure analysis where the software provides a significant part of the system functionality.
Human Reliability Assessment	Part C, Chapter 32;	This technique is not applied to software.

R+M Analysis Technique	Cross Reference	Software Applicability
R&M Aspects of Sub-contracts	Part C, Chapter 48;	This technique is not applied to software.
Data Reporting, Analysis and Corrective Action (DRACAS)	Part C, Chapter 18;	Current in service data collection systems generally fail to log adequate data in order to diagnose the cause of software failures. If software is to be the subject of an effective DRACAS then this should be considered from the outset of the programme and facility built into the system for self diagnosis and “black box” type data logging.
R&M Demonstration	Part C, Chapter 39, 40, 41;	Demonstrations should be agreed to include or exclude software failures. The reason for excluding software failures is that in practice they prove very difficult to assess chargeability and therefore costly to establish a rigorous demonstration method. Where excluded there should be robust argument for the acceptance of software based upon the evidence from other R&M techniques or previous experience.
In-Service Data Collection	Part C, Chapter 47;	Current in service data collection systems generally fail to log adequate data in order to diagnose the cause of software failures. If software is to be the subject of in service data collection then this should be considered from the outset of the programme and facility built into the system for self diagnosis and “black box” type data logging.

Table 3 Software Applicability in R&M Analyses

Model Name	Model Description
Jilinski and Moranda	This model calculates and estimates mean time to failure and pdf of time to find remaining errors, using as input, the number of errors in a debugging interval and the number of such intervals in the period of interest.
Shooman	The Shooman model assumes an exponential probability distribution based on CPU time. Its parameters include: • Total number of initial errors • Total number of machine language instructions • Total number of errors corrected at each debugging • Others to determine the probability of zero failure in a given operation period and MTTF
Brooks and Motley	This method provides a prediction of a program initial error content and error rate based on a multiple regression analysis of the performance of two large DoD software projects. The length of the programme was shown to be the biggest single prediction of error rate.
Goel and Okumoto	This model allows for the possibility that software errors are not removed with certainty. Classical and Bayesian statistical methods are used to estimate the parameters of the model to output Reliability, number of Errors, Time to next failure etc.

Table 4 Example of Software Reliability Models

LEAFLET B15/0

REFERENCES

1. Def Stan 00-42, Reliability and Maintainability Assurance Guidelines, Part 2, R&M Case.

2. IEC 61508 Functional safety of electrical/electronic/Programmable electronic safety related systems. Parts 1 to 7 International Electrotechnical Commission 1997.

3. Def Stan 00-55 Part 1, Requirements for Safety Related Software in Defence Equipment, Requirements.

4. Lyu, M.R., Handbook of Software Reliability Engineering, McGraw Hill,1995.

5. BS5760 Reliability of Systems, Equipments and Components, Part 8 : Guide to Assessment of Reliability of Systems containing Software, Draft for Approval July 1997.

6. Kan, S.H., Modelling and Software Development Quality, IBM Systems Journal,Vol 30, No. 3 1991.

7. Gaffney, J.E. and Davis, C.F., An approach to estimating Software Errors and Availability, Proceedings of the 11th Minnowbrook Workshop on Software Reliability July 1988.

8. Musa, J.D., Iannino, A., and Okumoto, K., Software Reliability, Measurement, Prediction, Application, McGraw Hill, 1987.

9. Ohba, M., Software Reliability Analysis Models, IBM Journal of Research and Development, Vol 21, No 4 1984.

10. Yamada, S. and Osaki, S., Software Reliability Growth Modelling: Models and Assumptions, IEEE Transactions on Software Engineering, Vol SE-11, No 12 1985.