

Reliability of systems, equipment and components

Part 8. Guide to assessment of reliability of systems containing software

ICS 21.020; 35.080

Confirmed December 2011

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW

Committees responsible for this British Standard

The preparation of this British Standard was entrusted to Technical Committee DS/1, Dependability and tetrotechnology, upon which the following bodies were represented:

- Association of Consulting Engineers
- Association of Insurance and Risk Managers (Airmic)
- Association of Project Managers
- British Railways Board
- British Telecommunications plc
- Centre for Software Reliability, City University
- Chartered Institution of Building Services Engineers
- Civil Aviation Authority
- Consumer Policy Committee of BSI
- Cranfield University
- Defence Manufacturers' Association
- Federation of the Electronics Industry
- GAMBICA (BEAMA Ltd.)
- Institute of Logistics
- Institute of Quality Assurance
- Institute of Risk Management
- Institute of Value Management
- Institution of Chemical Engineers
- Institution of Electrical Engineers
- Institution of Mechanical Engineers
- Institution of Plant Engineers
- London Underground Ltd.
- Ministry of Defence
- Railtrack
- Railway Industry Association
- Royal Institution of Chartered Surveyors
- Safety and Reliability Society
- Society of Environmental Engineers
- Society of Motor Manufacturers and Traders Limited
- United Kingdom Cals Industry Council
- West Midlands Enterprise Board

This British Standard, having been prepared under the direction of the Management Systems Sector Board, was published under the authority of the Standards Board and comes into effect on 15 October 1998

© BSI 1998

Amendments issued since publication

Amd. No.	Date	Text affected

The following BSI references relate to the work on this standard:
Committee reference DS/1
Draft for comment 96/402282 DC

ISBN 0 580 28207 4

Contents

	Page
Committees responsible	Inside front cover
Foreword	iii
0 Introduction	1
Guide	
1 Scope	1
2 Normative references	1
3 Definitions	1
4 Basic concepts	4
4.1 System reliability	4
4.2 Physical failure and design failure	4
4.3 Software failure	4
4.4 Measurement	7
4.5 Software reliability	8
5 Management overview	10
5.1 A management framework for software reliability assessment	10
5.2 Purposes of measurement	11
5.3 Data collection	11
5.4 Product-based software reliability assessment	11
5.5 Process-based software reliability assessment	12
5.6 Product models	13
5.7 Process models	13
5.8 Applicability and limitations of methods	14
5.9 Procedures	17
6 Software reliability assessment techniques	18
6.1 Classification of techniques	18
6.2 Software development process models	18
6.3 Software property models	22
6.4 Stochastic reliability models	24
6.5 Assessment of high reliability for software	48
7 Application procedures	49
7.1 Introduction	49
7.2 Procedures for use with process models	51
7.3 Procedures for use with product property models	52
7.4 Procedures for use with stochastic reliability models	52
7.5 Data collection forms	62
7.6 Logistics of software maintenance	63
Annexes	
A (informative) Forms used in data collection	68
B (informative) Mathematical descriptions of stochastic reliability models	71
C (informative) Predictive accuracy of stochastic reliability growth models	81
D (informative) Bibliography	85

	Page
Figures	
1 Mistake, fault, error, failure relationship	5
2 Software failure mechanism in a simple hierarchical system	6
3 Classification of stochastic reliability models	26
4 Fundamental reliability assessment problem: time to failure	29
5 Fundamental reliability assessment problem: failure count data	29
6 Example of failure history graphs and use of LCM	30
7 System failure due to activation of latent faults	34
8 Fault activation and correction in Jelinski-Moranda	35
9 Illustration of why the assumption of uniform fault size leads to optimistic estimates	36
10 Fault activation and correction in LSRG	38
11 Example of a u-plot: assessment of bias in predictions	44
12 Graphical notation for relationship database structure	58
13 Database structure: single product on single installation	58
14 Tables and attributes for single installation data	59
15 Database structure: multiple products on several installations	59
16 Tables and attributes: multiple products and installations	61
17 Interaction of support cost drivers	67
A.1 Form 1: incident report	68
A.2 Form 2: software item use log (calendar time)	69
A.3 Form 3: software item use log (usage time)	70
Table	
1 Management overview table	10
List of references	Inside back cover

Foreword

This Part of BS 5760 has been prepared by Technical Committee DS/1. It supersedes DD 198 : 1991, which is withdrawn.

This Part of BS 5760 describes some of the techniques available for assessing the reliability of systems containing software. It provides guidance to developers and procurers of such systems on how to apply some of the better established methods of assessment, and on which to avoid. The methods can be applied to any type of system, regardless of its intended function (although there are certain limitations in the case of high-integrity systems).

It is intended that these guidelines should be applied (in addition to any other necessary techniques) even for applications where extremely high reliability is required, in case the assessed level turns out to be inadequate.

Clause 4 identifies the basis on which this Part of BS 5760 is founded. It describes the fundamental concepts associated with software reliability and is intended to provide an easily understandable introduction for the non-specialist reader.

Clause 5 provides an overview of software reliability issues for those who need to understand the results of modelling software reliability and a non-technical summary of the available methods. It addresses the high-level issues associated with the measurement of software reliability. The different categories of model and the management of issues associated with their application are described. The relationship between reliability and integrity is introduced, and limits of the levels of reliability which can justifiably be claimed for software are discussed.

Clause 6 contains a more detailed technical description of the methods under the headings of process measurement (assessment of the quality of the software development process) and product measurement (assessment of the delivered software product).

Clause 7 contains a more detailed technical description of the procedures for application of the methods.

Annex A contains examples of forms used in data collection. Annex B contains mathematical descriptions of some of the better-known software reliability models. Annex C contains mathematical descriptions of some techniques which can be used to assess the accuracy of the predictions obtained from software reliability models, correct for bias in the estimates, and combine estimates obtained from using different models. Annex D contains a bibliography of the documents referred to in this Part of BS 5760. Numerals in square brackets throughout the text refer to items in the bibliography. Annexes A to D are informative.

Summary of pages

This document comprises a front cover, an inside front cover, pages i to iv, pages 1 to 88, an inside back cover and a back cover.

Introduction

Techniques for measuring and predicting the reliability of hardware are already widely applied. With the increasing use of computers there is a need to establish equivalent methods for evaluating the reliability of systems containing software.

The failure mechanism of software is not a physical process. A system containing software can fail when a latent fault within a software component is activated. Such faults are introduced by human error in the definition, design or development of the software, and are activated when particular circumstances are encountered during the operation of the system. Latent faults may also be present in the design of hardware, but are usually assumed to have been removed before the system is put into service, and are therefore discounted in the prediction of reliability. However in complex hardware, such as a microprocessor chip, design faults also contribute significantly to failure, and such complex designs pose similar problems of reliability assessment to those encountered with software.

This Part of BS 5760 describes the methods that are currently available for assessing the reliability of systems with respect to failures due to software faults. Many of these methods can also be used to assess system reliability with respect to the activation of design faults in hardware. Suppliers and users need to be able to specify and measure the reliability of all kinds of systems containing software ranging from commercial billing systems to automotive electronic control units, nuclear reactor control systems and computer controlled missiles. In some cases the probability of failure of such systems due to software faults is of greater concern than the probability of their failure due to physical causes. An assessment of the total reliability of any such system cannot afford to ignore the effect of latent design faults.

This British Standard provides a structure within which software reliability assessment issues can be addressed from the early stages of system requirements definition, through design, development and testing, until the actual reliability can be assessed during system trial and operation. The setting of achievable reliability targets and the prediction of reliability in the early phases of development requires the use of expert judgement based on experience and historical data. The monitoring of the ongoing development process and the assessment of the level of reliability achieved requires careful measurement. All of the relevant data should be collected and then analysed using statistical methods.

Guidance is given on all of these aspects of the prediction and assessment of system reliability with respect to the manifestation of software faults.

Guide

1 Scope

This Part of BS 5760 gives guidance on the assessment of reliability of systems containing software with respect to those failures that are due to the activation under certain environmental circumstances of latent design faults located in software items.

NOTE. Latent software design faults are due to human error during the definition, design and development phases of system production.

Guidance is provided on the assessment of system reliability both by assessment of the product (the software) and by assessment of the process (the means by which the software is developed). This guidance applies to any system containing software regardless of its intended function. There are limits to the level of reliability that can be assessed quantitatively.

This Part of BS 5760 seeks to classify some of the more established methods and to provide guidance to the practitioner in applying them.

A bibliography is provided in annex D.

2 Normative references

This Part of BS 5760 incorporates, by dated or undated reference, provisions from other publications. These normative references are made at the appropriate places in the text and the cited publications are listed on the inside back cover. For dated references, only the edition cited applies; any subsequent amendments to, or revisions of the cited publications apply only when incorporated in the reference by amendment or revision. For undated references, the latest edition of the cited publication applies, together with any amendments.

3 Definitions

NOTE. These definitions cover the field of software reliability measurement. They have been made as consistent as possible with the terms employed in general reliability, availability and maintainability work, and in particular with the definitions in BS 4778. In some cases it has been necessary to extend or modify the BS 4778 definitions slightly, since it does not take into account some aspects of systems containing software. Where this has been necessary, the BS 4778 definition is quoted with a note or amendment (see for example 3.11).

3.1 activation (of a fault)

The event in which a latent fault gives rise to a failure in response to a trigger.

NOTE. Also referred to as 'manifestation of the fault'.

3.2 attribute

Any observable property of an entity.

3.3 baseline

A major version of a system selected for release to customers and/or for the purpose of measuring some attribute, e.g. reliability.

3.4 bug

Synonymous with design fault, usually in software.

3.5 calendar time

Time as commonly recorded by clocks and proportional to the rotation of the Earth.

NOTE 1. Also known as elapsed time, or real time, or wall-clock time.

NOTE 2. This time is 'public' in the sense that all observers can agree on it, except that geographical time zones may need to be taken into account.

NOTE 3. Reliability measurement usually requires the use of 'operating time', which is a measure of the total time during which a defined sample of systems has been in use or on trial. Operating time is generally not the same as real time.

NOTE 4. Measurement of software reliability requires a measure of execution time, which is the operating time of a software item.

3.6 direct measurement

Measurement which can be made by empirical observation of a single attribute and does not depend on the measurement of other attributes.

3.7 entity

Any object, event or process in the real world.

3.8 execution profile

A measurement of the proportion of total execution time that is spent executing code within each subsystem or module of a software item.

3.9 execution time

A measure of the amount of execution undergone by a software item.

NOTE. The measure chosen will depend on the type of system. Generally, it will not be equivalent to real time. Possible measures are processor time consumed, number of instructions executed, etc.

3.10 external attribute

An attribute of a system which characterizes its interaction with its environment.

3.11 failure

The event of an item ceasing to perform a required function or provide a required service in full or in part.

NOTE 1. The term 'item' may refer to a complex system, consisting of hardware, software, or both.

NOTE 2. A failure is an event in time. A fault is a state of the system.

NOTE 3. A failure may be due to physical failure of a hardware component, activation of a latent design fault or an external failure.

NOTE 4. Following a failure, an item may recover and resume its required service after a break, partially recover and continue to provide some of its required functions (fail degraded) or it may remain down (complete failure) until repaired.

NOTE 5. The definition of failure in BS 4778 is inadequate for the purposes of this standard since it does not explicitly allow for transient failures. Also the notes to the definition may be interpreted to imply that the definition excludes events due to the activation of pre-existing latent design faults and in particular events due to the activation of software faults.

NOTE 6. The definition used in this standard is consistent with the definition in BS 4778 but addresses the inadequacies described in Note 4 above (see 4.2 and 4.3.1).

3.12 failure mode

The effect by which a failure is observed.

NOTE. This definition is identical with 14.5.4 of BS 4778 : Section 3.1 : 1991.

3.13 failure severity

The seriousness of the effect of a failure.

3.14 incident

An event during operation of an item which may indicate that a failure has occurred.

3.15 index of merit

Non-dimensional value used to compare the reliability of two or more systems.

3.16 inspection

The comparison of the output products of a development phase with the input products in order to ensure that the former are a correct transformation of the latter.

NOTE 1. The purpose is to detect and correct instances of non-conformance (see 3.23) before they become faults in the delivered system.

NOTE 2. The technique is an example of verification (see 3.37).

NOTE 3. A particularly formal and thorough method of inspection in common use is that known as Fagan inspection, after its inventor M.E. Fagan.

NOTE 4. Measurements of instances of non-conformance detected by inspection can be used as indicators of quality of delivered software (e.g. defect density, see 6.2.2.3).

3.17 installation

A single hardware machine capable of running one or more instances of a software product.

3.18 instance

A single copy of a software product in operation or on test on a single installation.

3.19 measurement

The process of empirical, objective assignment of numbers (or symbols) to properties of entities (objects and events) in the real world in such a way as to describe them.

NOTE 1. A measure is a defined mapping of the entities in the real world onto a scale of numbers or symbols. In order for a measure to be meaningful, the relationships among the entities should be represented by corresponding relationships among the associated numbers or symbols (see 4.4).

NOTE 2. Measurement is the activity of applying a measure to a property of an entity in the real world by establishing its actual value for an entity.

3.20 modification (change)

Alteration to the design of an item.

NOTE 1. Changes may be made for reasons of corrective, adaptive or perfective maintenance.

NOTE 2. The definition in 191-01-13 of BS 4778 : Section 3.2 : 1991 is 'The combination of all technical and administrative actions intended to change an item'.

3.21 modification state

A specification of which modifications have been carried out on a particular example of an item in use on a given installation.

NOTE 1. The item may be a piece of hardware or an instance of a software component.

NOTE 2. The modification state should specify both the baseline version of the item, and all minor modifications that have been carried out.

3.22 module

A self-contained software item with a specified function and a defined interface to the rest of the system.

3.23 non-conformance

Incorrect, incomplete or superfluous implementation of an input product by an output product at some phase of system development.

NOTE. Among practitioners of inspections the term 'defect' is used synonymously. However 'defect' is deprecated since it might have legal connotations in some contexts and will not be used in this British Standard without qualification.

3.24 operational profile (usage)

A characterization of the conditions of use of a system.

NOTE 1. The term is generally applied to software. The profile can sometimes be defined by a partitioning of the input space and estimate of the probability of encountering an input from each class. For highly complex systems, e.g. computer operating systems, this may not be possible, and a less precise characterization should be used, such as the 'mix' of types of job being processed.

NOTE 2. Definition of conditions of use is essential for measurement of reliability (and other dependability attributes), since a system will generally exhibit different levels of reliability under different conditions. A system trial should therefore be performed using a realistic operational profile.

3.25 random test (statistical test)

The strategy of selecting test cases at random according to the probability with which they are expected to be encountered in operation, in order to ensure that the operational profile used in test and trial is a reasonable approximation to reality.

3.26 real time software

Software that has to return an output within a certain time interval, in order to be able to affect some social or physical process.

3.27 safety

The freedom from unacceptable risks of personal harm.

NOTE 1. Safety is defined in the context of risk of personal harm. It is traceable quantitatively in decision-making on acceptable risks.

NOTE 2. This definition is identical with 11.2.1 of BS 4778 : Section 3.1 : 1991.

3.28 software

Computer program code and its associated data, documentation and operational procedures.

3.29 software failure

System failure due to the activation of a design fault in a software component.

NOTE 1. All software failures are design failures, since software consists solely of design, and does not wear out or suffer from physical failure.

NOTE 2. Since the triggers that activate software faults are encountered at random during system operation, software failures also occur randomly.

3.30 software fault

A design fault located in a software component.

3.31 software fault-tolerance

A design feature of software that enables error recovery to be performed after the manifestation of a software fault.

3.32 software reliability

The probability that no latent fault in a software component of a system will be activated during a given time interval in the operation of the system under given conditions of use.

3.33 specification fault

A fault of an item that results from a required function having been incorrectly or incompletely defined.

NOTE. Specification faults often give rise to usability problems in operation, but can lead to other types of incident also. They can only be detected by validation, not verification.

3.34 trial

Exercising of a system under conditions as close as possible to the expected operating conditions.

NOTE 1. This is necessary to measure dependability attributes such as reliability.

NOTE 2. A trial may require some form of random testing (see 3.25).

3.35 trigger

The particular combination of circumstances that activates a latent fault.

NOTE 1. Most design faults remain latent for a long time.

NOTE 2. The term applies particularly to software faults. (All software faults are design faults.)

NOTE 3. The circumstances include selection of particular inputs together with a certain internal state of the system.

NOTE 4. In the case of software, the internal state is defined by the module being executed, the execution point reached and the values of internal variables and pointers.

3.36 validation

Steps taken to ensure that a system meets the requirements of the user.

NOTE 1. Validation answers the question: 'Are we building the correct system?'

NOTE 2. Validation may require a trial of some kind.

NOTE 3. See 2.18 of BS EN ISO 8402 : 1995

3.37 verification

Steps taken to ensure that the output products of any development phase are a correct transformation of the input products.

NOTE 1. Verification answers the question: 'Are we building the system correctly?'

NOTE 2. Verification includes such activities as inspection, proof of correctness, etc.

NOTE 3. See 2.17 of BS EN ISO 8402 : 1995

4 Basic concepts

4.1 System reliability

When considering a system, all factors (software, electronic, mechanical and human) likely to influence the reliability of that system have to be addressed. Certainly, it is not possible to generate a system consisting solely of software. Fortunately many of the fundamental tools, such as reliability block diagrams, fault tree analysis, reliability growth monitoring etc. which are applied at the system level can accommodate these diverse factors. Reliability techniques and their impact on hardware, software and human interfaces are given within BS 5760 : Part 2.

Advances in component miniaturization and improved component reliability have resulted in systems becoming more and more complex. To maximize the benefits that these improvements in technology can provide, greater emphasis is being placed upon the design process. Currently, reliability predictions are very much concerned with the elimination of design faults as early in the design process as possible. Metrics (software measures), for example, can be applied equally to a hardware design as they can to a software design.

In the case of a highly complex system, the system is usually subdivided into simple modules that can be readily understood by individual designers, although the ensuing interface specifications can be highly complex. For very simple modules within the complex structure, there is a tendency for the rigorous design controls to be relaxed. However, experience has shown that these simple modules can sometimes cause the greatest problems, the degree of care taken to eliminate design faults is a major factor that determines the reliability of the system.

Software and hardware faults can cause system failure directly but it can also be caused indirectly as shown below:

- software faults that generate software faults that cause system failure;
- software faults that generate hardware faults that cause system failure;
- hardware faults that generate hardware faults that cause system failure.

The chain reaction effect such as those given in the above examples can be difficult to detect. Consider the case of a power management subsystem (hardware or software controlled) that incorrectly permits an occasional small voltage spike down the power line. However, the spike is insufficient to cause a component to immediately fail, but has a weakening effect. After about a hundred spikes the component fails. In these cases, the component is replaced and the root cause usually remains undetected until either the frequency of spikes increases for some reason or a less robust component is substituted in the design.

It should always be remembered that the user of a system is concerned with the adequacy of performance, the frequency of failure and the ease and cost of repair. Whether the failure is due to software, hardware or even a combination of them both is, from the user's viewpoint, completely irrelevant.

4.2 Physical failure and design failure

Complex systems can fail for two fundamentally different reasons.

a) Physical failure: a hardware component fails, for example a resistor 'shorts', or a logic gate 'sticks'. After its individual failure, the component is faulty; there is a fault in the system. Repair consists of replacement of the faulty component to restore the system to its previous functioning state. After repair the system should continue to function and should not necessarily fail again on encountering the same circumstances as those that led to the previous failure. Failures of this kind are often called from a statistical viewpoint 'random failures'.

b) Design failure: a fault in the design of the system is activated in response to certain conditions. The fault may have been present for some time, although latent. Repair consists of a modification to the design of the system to remove the fault. Although such 'corrective maintenance' may introduce new faults, it generally improves the design (and increases the reliability) of the system. This kind of failure is often called 'systematic' because unless the design is changed to remove the fault, the same failure will recur if the same circumstances arise.

Failure in manufacture is normally attributable to hardware and will not therefore be considered in the context of software. A software failure is a system design failure due to a fault that is located in a software component. Since software is a part of the design of the system (although once compiled and loaded, it has a physical representation), it can only undergo design failure. Many failures of modern complex digital systems are associated with software failures.

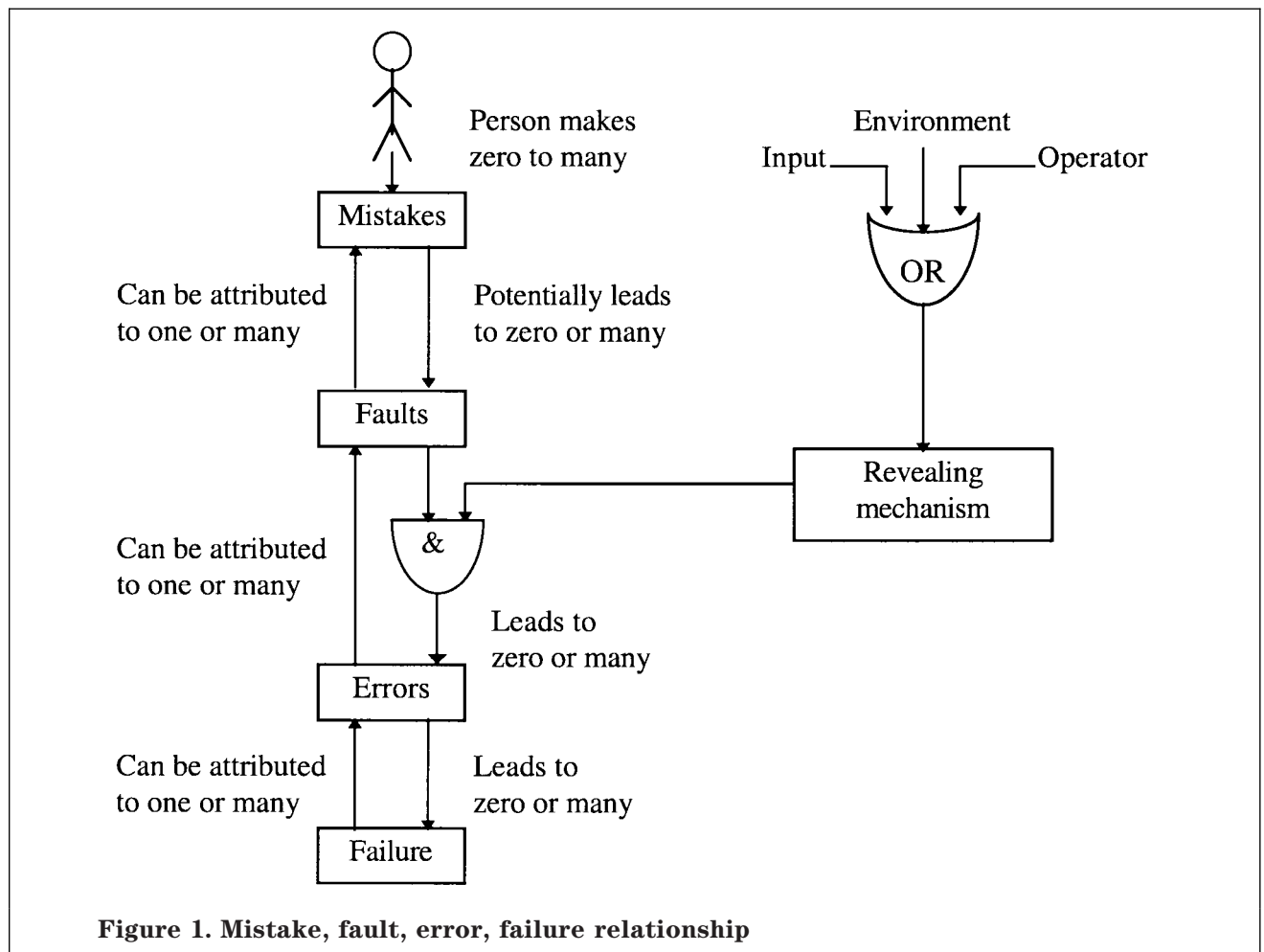
4.3 Software failure

4.3.1 Usage of terms

The relationship between mistakes, faults, errors and failures, even the meaning of the terms in relation to software, is a matter on which there is apparent inconsistency in existing standards. As a result it is essential that the way in which the terms are used in this British Standard is clearly understood.

This Part of BS 5760 uses definitions for mistake, fault, error and failure which correspond with those used in BS 4778, though they are not all identical. This Part of BS 5760 takes the view that these definitions are applicable to software, provided that they are interpreted appropriately.

Figure 1 illustrates the relationship between mistake, fault, error and failure based on these definitions.



4.3.2 Mistake, fault, error, failure

This Part of BS 5760 uses the definitions of mistake, fault and error given in BS 4778 and the definition of failure given in 3.11.

A mistake (i.e. human error) during some phase of software development can lead to a non-conformance between the input product and output product of that phase. For example a mistake during the coding phase can lead to source code which is not a correct implementation of the detailed design specification. Unless such a non-conformance is detected (e.g. by inspection) and removed, it can be propagated during the succeeding phases and eventually lead to one or more faults in the delivered software. (The same applies to the creation of latent design faults in hardware.)

A software fault remains latent until a particular combination of inputs, operator actions, other environmental circumstances and internal states referred to as the trigger coincide during test, trial or operation and activate the fault. The immediate effect is a local error within the component containing the fault, i.e. a discrepancy between its actual internal state and the 'correct' state. If this propagates across the component interface then that

component will have failed and a fault at a higher level will now exist within the system leading to a more widespread error, and so on until a failure occurs at the system interface, i.e. the system ceases to behave as required.

This is illustrated in figure 2 for a simple system structured hierarchically in three levels so that components at the higher levels invoke those at lower levels in order to obtain some required service. The following list which corresponds to figure 2 details the software failure mechanism.

- The system S invokes subsystem SS1 which in turn invokes module M1.2 (S constitutes the 'environment' of SS1 and SS2. SS1 in turn constitutes the environment of M1.1, M1.2 and M1.3). S is processing input from its environment. When S invokes SS1 it passes down certain information whose nature depends on its input and internal state. SS1 in turn passes to M1.2 information which depends on that received from S. M1.2 is therefore being invoked in a particular way which depends ultimately on the environment of the overall system S. Suppose now that M1.2 contains a latent fault which is activated under the particular set of circumstances that it now experiences.

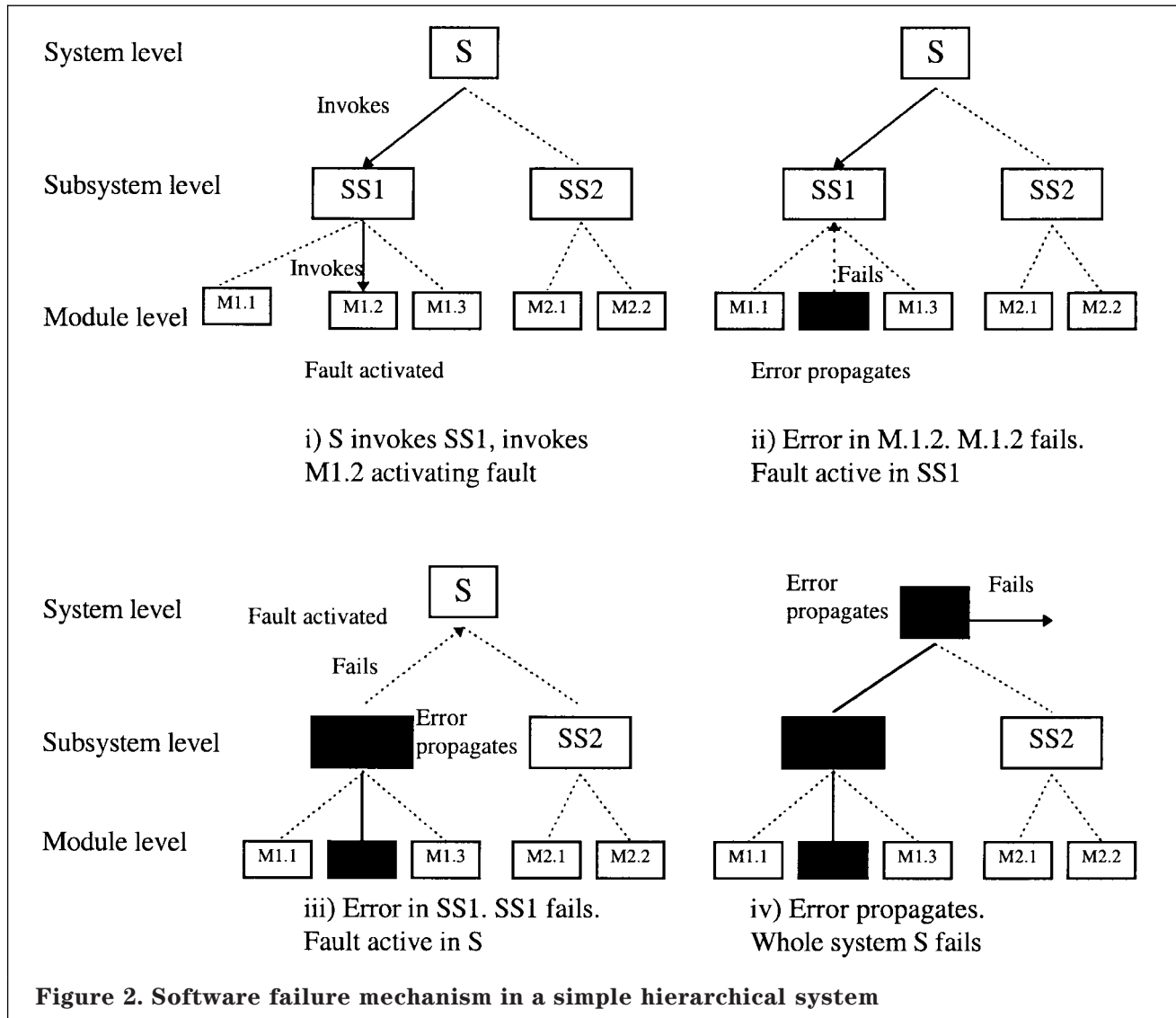


Figure 2. Software failure mechanism in a simple hierarchical system

b) The activation of the fault gives rise to a local error within M1.2 which is therefore unable to perform its required function on behalf of SS1. M1.2 has now failed and this constitutes an active fault within SS1.

c) This active fault in SS1 now leads to an error in the internal state of SS1 so that it in turn cannot perform its required function on behalf of S. SS1 has now failed and an active fault now exists at the level of the system S.

d) Finally an error propagates at the system level and S is no longer able to perform all of its required functions. A failure occurs at the interface between the system S and its environment. This failure can be recognized by a certain failure mode or set of observable symptoms. It may also have a detrimental effect on its environment (including human users and other systems which depend on its service) and its severity is a measure of the magnitude of its effect.

Figure 2 assumes that the system is not fault tolerant. It might be possible to design a system so that it can detect and contain internal errors and recover from them automatically, so preventing a system level failure although local failures of one or more components might occur.

If the system S encounters the same operational circumstances again a similar failure will recur. In order to prevent this it is necessary to carry out a modification to remove the fault from module M1.2. This involves following the causal chain of 'fault-error-failure' backwards in order to diagnose the nature and location of the fault. The process of fault diagnosis and removal is referred to as corrective maintenance or debugging.

A simple example has been chosen for purposes of illustration. Any real system is likely to be much more complex and might not be structured in a straightforward hierarchy. However, the same principles apply.

4.3.3 Nature of software failure

Software failures generally have the following characteristics.

- a) They are due to latent design faults in software components of the system. These design faults are caused by human error during development or maintenance of the system. A failure occurs when a fault is activated by a particular set of operational circumstances, referred to as the trigger for that fault.
- b) They are transient. If the trigger is removed the system can recover and resume normal service. Often it is possible to reload the software and restart the system following a failure.
- c) They are systematic since until the latent fault is removed by corrective maintenance the system will fail again in a similar mode whenever the trigger is encountered.
- d) They are random since the trigger for each fault is encountered at random (see 4.3.4).
- e) They tend to be infrequent since the trigger is usually a very rare combination of operating circumstances. (Faults which are activated very frequently would be likely to be detected and removed in early testing.)
- f) Their modes and effects tend to be unpredictable so that it is difficult to provide fault-tolerant design features or safety devices to guard against them automatically.
- g) Although they tend to be infrequent their consequences can be catastrophic.

Failures due to faults in hardware design share many of these characteristics.

4.3.4 Random software failures

Software failures are systematic because they can be reproduced at will by replicating the trigger. (Debugging frequently involves reproducing a failure condition by deliberately subjecting the system to the same operating conditions that were established when a failure was observed during operation.) Systematic failures are often considered to be purely deterministic.

Physical hardware failures generally occur at random and hardware reliability can therefore be measured using a probabilistic or stochastic approach, e.g. by estimating a mean time to failure (MTTF) or failure rate for the stochastic process of failure. Software reliability can be measured in the same way only if software failures can be considered to occur randomly.

There are two main approaches to describing probability.

In the frequentist approach the probability of an event is defined as the limit of the frequency of its occurrence as the number of identical experiments of which the event might be an outcome increases without limit. For example, as the number of tosses of a fair coin increases the proportion of heads should approach 0.5, which is the probability that a given toss will result in a head.

In the Bayesian approach the probability of an event is defined as a measure of the observer's uncertainty that it will occur. The probability assigned to an event depends on the observer's knowledge and might change as the observer gains information from further observation. This approach allows a probability to be assigned to a unique event, e.g. the probability of a given individual dying within a certain time can be estimated for the purpose of setting a life assurance premium.

Either approach may be used to describe hardware reliability.

Activation of any given software fault is a unique event in the operational life of the system. Therefore in order to describe software failure as a random process the Bayesian approach is appropriate.

A software failure mode can only be reproduced once the trigger for the corresponding fault is known. The number and nature of the software faults in a system in operation are generally not known and their triggers will be encountered at random with a certain frequency which depends on the environment. Therefore the process of activation of software faults is also random and stochastic methods can be applied in order to measure software reliability [2].

4.4 Measurement

4.4.1 General

A development or manufacturing process is a set of activities which takes place over time, consumes resources and generates a product. In order to manage development or manufacture it is necessary to measure the process itself, the expenditure of resources and the quality of the delivered product (see 5.2).

4.4.2 Fundamental measurement theory

The following is a very brief and simplified summary of fundamental measurement theory as applied to systems containing software [3] (although similar considerations apply to the development or manufacture of other types of equipment).

Measurement makes observation more precise as follows.

- a) Measurement assigns a number to each entity depending on the degree to which it possesses the attribute. The 'number system' used in this mapping may be real numbers, integers or a set of labels of categories.
- b) The representation condition requires that relationships between entities, determined by the attribute, are modelled by corresponding relationships among the numbers assigned. If this holds then the assignment of numbers to entities can constitute a measure of the attribute and the number system used is referred to as a scale.
- c) The following types of scale are commonly used and all occur in connection with software reliability assessment:
 - 1) nominal: entities are assigned to one of a set of labelled categories, e.g. classification of software faults as 'incorrect logic', 'uninitialized variable', etc.;
 - 2) ordinal: entities are assigned to one of a set of ordered categories, e.g. classification of failures as 'critical', 'major', or 'minor' based on the attribute 'severity';
 - 3) interval: a real number is assigned to each entity but both the units and origin are arbitrary, e.g. calendar time can be measured in years since 00.00 hours on 1 January in the year 0 Common Era (CE) or in days since the start of a project;
 - 4) ratio: a real number is assigned to each entity with arbitrary units but a fixed origin, e.g. testing time can be measured in operating hours or CPU seconds but the start of testing always corresponds to 0 testing time;
 - 5) absolute: counting of entities, e.g. counting the number of faults activated during a given period of testing.

4.4.3 Entities

Meaningful measurement requires the selection of appropriate entities, the clear definition of attributes in such a way that they can be quantified and the choice of appropriate scales such that the representation condition holds. Not every assignment of numbers to entities constitutes a meaningful measure.

Entities that can be measured in software development (or in other production or manufacture) can be classified as one of the following three types.

- a) Product: output of a process, e.g. specification document, source code, delivered system.
- b) Process: activity which consumes resource and generates product, e.g. writing specification, coding, system testing.
- c) Resource: something of value consumed by a process, e.g. effort.

4.4.4 Attributes

4.4.4.1 Classification of attributes

Attributes can be classified as one of the following two types.

- a) Internal attributes: describe the entity in isolation, e.g. size (attribute) of a software component (entity) might be measured by 'number of lines of source code' (measure).
- b) External attributes: describe the interaction of the entity with its environment, e.g. reliability (attribute) of a system (entity) could be measured by 'average number of failures per operating hour' (measure).

In most cases internal attributes can be observed for a system without the need for it to operate and are said to be static, whereas external attributes can only be observed for a system in operation and are said to be dynamic.

4.4.4.2 Measurement of attributes

Measurements can be made at any of the following three levels.

- a) Raw data collection is immediate observation, e.g. completion of incident report, recording hours worked on job sheet.
- b) Direct measurements might require the extraction of refined data from raw data but do not depend upon the measurement of any other attribute, e.g. counting incident reports, totalling hours worked.
- c) Indirect measurements are derived from other measures by calculation or analysis, e.g. failure rate (number of failures/running time), productivity (number of lines of code/hours worked).

Where a measure of one attribute is found to be correlated with that of a different attribute, the first is referred to as an indicator of the second. To establish the predictive value of an indicator it is first necessary to measure both attributes independently in a sample of entities and deduce the degree of correlation. The sample should be both large enough and sufficiently representative to establish the correlation with a high level of confidence.

4.5 Software reliability

4.5.1 Using the concepts underlying system reliability (see 4.1), physical failure and design failure (see 4.2), software failure (see 4.3), and measurement (see 4.4), the following summarizes the basic concepts of software reliability and its assessment.

In simple hardware systems, it is possible to eliminate virtually all design faults prior to the working life of the system. Most methods of hardware reliability prediction therefore ignore design faults as a source of unreliability. However, software is almost inevitably complex (as are the processes of designing and developing it) and may dominate the contribution of physical failure

4.5.2 The meaning of software reliability can best be understood from the viewpoint of a user. Having taken the system containing software into service, the user may observe later that it does not perform as required. If the precise condition that reveals such unexpected performance had existed as a test case, then the software could have been corrected. Since it is not possible during testing to reproduce all the conditions that may be experienced during a lifetime of use, there are likely to be occasions when the system does not perform as required.

Users, suppliers and maintenance organizations are interested in methods of predicting the likelihood of such unexpected performance. The user's criteria for deciding whether system performance is not acceptable may well influence the prediction method used and the nature of the data input to it. The classification of performance into these categories should be made on an individual basis since a major 'fault' to one user may be merely a minor nuisance to another. A software fault that causes the system to stop functioning may be considered acceptable by one user (providing the cause is known) but to another user running a system controlling an industrial process, the same fault may be a severe embarrassment.

4.5.3 Faults in documentation (that could affect speed of modification or repair) may also affect users to different degrees. In some systems, certain parts of the software may be used more intensively than others and the user may wish to focus interest on those areas in which faults are most likely to be present. The importance of a fault to a user will be governed by such factors as the accessibility of the software for correction and the severity and immediacy of the consequences. The following list gives examples of applications where such factors are important:

- a) software unavailable for correction, e.g. guided missiles, computer microcode;
- b) software not readily available for correction, e.g. washing machines;
- c) software which could mislead users, e.g. teaching software for safety critical systems;
- d) systems in which software failure could lead to major malfunction, e.g. railway switching systems;
- e) software which has to function when required, e.g. nuclear reactor shutdown.

NOTE. In order to cater for various users it may be appropriate when delivering the results of a reliability assessment to place the various types or areas of failure into distinct categories so that each user can classify the significance and severity as appropriate to the case.

4.5.4 The reliability of a system [4] is the probability that it will operate without failure for a given period of time under given conditions of use. It is an external attribute of the product, and should be measured by observing the system in use or under test in an environment that closely approximates to those conditions of use that will be experienced in

service (referred to as a trial).

Measures of reliability include failure rate, mean time to next failure and probability of successful operation during a given period. Such measures are defined on ratio scales. They are indirect measures and should be derived from statistical analysis of direct measures of the occurrence of failure over operating time (either times between individual failures or counts of failures and accumulated operating time in successive periods). These in turn should be extracted from raw data consisting of records of each failure and operating time logged in some convenient way.

Where design failure is concerned, reliability growth is normally observed as the system is progressively modified to remove the design faults which give rise to failure, so that its reliability improves. Software failure is a particular category of design failure, in which the underlying design fault is located in a software module or interface. Normally records of second and subsequent failures due to a fault which has previously been activated are removed from the raw data and the assessment of reliability is based solely on the records of the first activation of each fault. The two following important points should be noted regarding the assessment of software reliability based on failure data.

- a) The accuracy of the estimates depends crucially on the extent to which the environment used during trial is representative of the environment in service. The same software product in different environments may exhibit very different levels of reliability. An environment may be characterized by an operational profile or set of probabilities of encountering various classes of input. This governs the probability of encountering the trigger for any given fault and hence its rate of activation (see 6.4.16).
- b) It is not possible to assess a very high level of reliability based solely on failures observed during a trial of reasonable length. However if there are no failures, the trial duration is insufficient to confirm, with a reasonable level of confidence, the required reliability. Increasing the trial duration (in some cases to several years) would be impractical from a cost and programme viewpoint.

Measures of certain process attributes and internal product attributes might correlate with the level of reliability observed in operation. These may be used as indicators to predict reliability during early development before a complete system is available for trial. However, evaluating such indicators is not equivalent to measuring reliability, and their predictive ability should be independently assessed. Examples of types of indicator are process characteristics (see 6.2), inspection statistics (see 6.2.2), and software properties (see 6.3).

4.5.5 The procedures outlined in this British Standard deal exclusively with the contributions to unreliability of failures due to the activation of latent faults in software. These in turn are due to mistakes in its design and development. The contribution to unreliability of physical failure of components is covered by BS 5760 : Part 2. In order to predict the reliability of the complete system, the estimate of unreliability due to faults in software design should be combined with the results of applying the methods described in BS 5760 : Part 2. Care should be taken to employ compatible measures of reliability for the two factors and to ensure that they are correctly combined.

5 Management overview

5.1 A management framework for software reliability assessment

There are reliability assessment needs at all stages of the software life cycle (see table 1). The purpose and nature of the assessment methods which can be used vary from stage to stage, but all contribute towards building confidence in the level of reliability that can be claimed for the final system. The aim should be to build confidence in the reliability as the software is being developed: it is unwise to wait until the end of development to discover that the level of reliability achieved is inadequate.

People in several roles (e.g. user, customer, certification agency, reliability manager, business manager, programmers and other technical staff, quality manager and project manager) can be expected to take an interest in reliability assessments of a software based system.

To satisfy the needs of these roles, software reliability assessment should be carried out at all stages of the life cycle, in parallel with and often based on existing verification and validation activities. The degree of precision and quantification associated with these assessments varies a great deal through the development cycle. The success of early software reliability assessment is dependent on finding ways to make visible what is happening within the development team; all too often, the inherently abstract nature of software and its development process serves to obscure the reliability of the developing product.

In the earliest stages of development, reliability assessment can be carried out mainly by studying the development process since there are only intermediate products to evaluate, and no final software of which to measure the reliability. Later, assessment can also be end product based, making use of static properties of the software, and of its dynamic failure behaviour. The aim of all these assessments is to build confidence by ensuring that a development strategy appropriate to a given achievable target is properly carried out, and by examining the end product and its behaviour when executed.

Sometimes, assessment needs to be applied to systems that already exist, and the need is for retrospective reliability assessment. This situation should be avoided if possible, as the benefits of carrying out assessment in parallel with development (e.g. earlier knowledge of problem areas) will be lost. Where retrospective assessment is unavoidable, the underlying principles still apply, but some of the detail may need to be adapted as appropriate.

Table 1. Management overview table

Subclause in this Part of BS 5760	Subclause title	Software life cycle phase				
		Definition/feasibility	Design and development	Implementation	Installation and commission	Operation and maintenance
5.1	A management framework for software reliability assessment	A	A	A	A	A
5.2	Purposes of measurement	—	A	A	A	A
5.3	Data collection	—	L	A	A	A
5.4	Product-based software reliability assessment	—	L	A	A	A
5.5	Process-based software reliability assessment	L	A	A	L	A
5.6	Product models	L	A	A	A	A
5.7	Process models	L	L	L	L	L
5.8	Applicability and limitations of methods	A	A	A	A	A
5.9	Procedures	L	A	A	A	A

NOTE. A = applicable; L = limited applicability.

5.2 Purposes of measurement

5.2.1 Assessment and certification

Before a product is delivered to the customer the developer should ascertain that it is fit for its intended purpose. This applies to systems containing software just as to other types of product. Certain quality goals are normally included in (or implied by) the requirement specification and may be stated in the development contract.

The developer should measure the system during a trial to ensure that it meets the reliability goals. The goals should be defined quantitatively so that conformance to the requirements can be established without doubt.

For a system containing software, the contribution to unreliability of the software components should be measured in addition to other factors. If it is found that the system does not (or will not) achieve its reliability goals due to unreliable software then measurement of the reliability actually achieved is useful in determining what action to take, e.g. estimating the further testing time required to reach the goal.

Regulatory requirements for certain levels of reliability may be imposed on the developer by external agencies, and these will often be legal obligations. This is particularly the case with safety-critical applications, such as nuclear reactor control systems and avionics systems on board aircraft. In such cases, the regulatory authority will often independently review measurements made during software development before awarding a certificate.

5.2.2 Maintenance cost estimation

After delivery software usually requires maintenance. Faults found in operation need to be corrected and the software may need to be enhanced or adapted to new environments. Measurements of reliability and maintainability will form the basis of all commercial decisions regarding maintenance contracts, warranty periods, estimation of size of support teams, etc. This is usually the case with large commercial applications, operating systems and similar software.

5.2.3 Improvement of the development process

Having measured the reliability of the software produced using particular development methods, and compared this to the levels of reliability achieved using different methods, the developer can assess the relative effectiveness of the various methods used, and adopt the better ones for future projects.

5.2.4 Control of the project

In addition to knowing if the project is staying within its constraints regarding budget and schedule, the developer should also take steps to find out if the system is likely to be adequately reliable when delivered. This requires the measurement of such things as non-conformances found in inspection and faults found in early testing which might be indicators of the eventual reliability of the delivered system.

5.2.5 Estimation of future projects

Estimates of the reliability of the delivered system that can be expected from any proposed new development should be based on experience of previous projects. This is only possible if measurements are made of the levels of reliability of earlier systems in service. This constitutes part of the 'corporate memory' of the development organization, along with records of the cost and schedule and development methods employed.

5.3 Data collection

The gathering of evidence about software reliability implies data collection and storage, and the varying nature of the through-life activity implies wide diversity in the kind of data that is to be used. Most organizations will benefit from harmonizing these data collection procedures, both across projects, and with other related activities like configuration management and change control: some data needed for software reliability assessment purposes (e.g. fault reports, version numbers) have their natural origin in such existing systems.

5.4 Product-based software reliability assessment

5.4.1 General

Various product-based methods may be used to assess the contribution of software to system reliability; these make use of three main types of information: software properties, fault data, and failure data. Most current work on reliability assessment uses fault or failure data, but there are some methods which use software properties and a few which combine the various types of data.

The main features of these approaches are outlined in 5.4.2 to 5.4.4.

5.4.2 Software properties

Assessment based on software product properties analyses the form, structure, content and complexity of the software itself (and other kinds of intermediate and final products of the development process) for consistency with the targeted level of reliability. In particular, knowledge of the software and system structure together with estimates of the reliability of individual parts (possibly gained from previous experience in service, for example in cases where the parts have been reused) can be used to produce a combined assessment of total software and system reliability.

These methods often form part of a more general quality assurance activity, intended to predict and assess quality factors other than reliability such as maintainability, portability, etc. They depend upon various measures, which are measurable attributes of the product, such as number of lines of source code, decision points, operators, or faults found in inspections.

Some of the models are intended to predict reliability, etc. at an early stage of the life cycle, such as requirements specification or high level design. In order to be used predictively, the relationship between the observed metric values and the achieved level of reliability needs to be established statistically with reference to preceding similar products. This gives rise to the following two main problems:

- a) deciding the criteria to be used to judge that an earlier product is similar to the current one;
- b) the practical problem that a large amount of data needs to be available from earlier developments.

5.4.3 Early fault data

Fault data, like software property data, can be available early in software development, and has been found useful by many organizations for predicting the fault and failure levels likely to be experienced later in development, and in the use of the software following delivery. For example, an abnormally high number of faults found in design review of a software component could result from a complex design, which could in turn lead to further problems at later stages.

Organizations which have taken the trouble to collect and analyse data about faults discovered at various development stages, and data about achieved reliability levels, have derived relationships which help them in predicting not only fault and failure frequencies, but in allocating resources to (for example) testing activities, in the light of likely fault levels.

Unfortunately, there are no 'off-the-shelf' solutions, enabling predictions on the basis of generic models. Software engineering is still at the stage where each organization's processes are unique to that organization, and even within the organization, 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. Nonetheless, the value and effectiveness of such approaches is increasingly seen as worth the investment.

5.4.4 Failure data

Once the software has reached the stage of being executable, statistical methods can be used to assess current reliability and predict future reliability growth from records of system failure and the extent of use of the system. This should be done both during a trial (a period of testing in a realistic environment after system integration) and also in service.

Failure data methods can only be used when a system already exists and is exhibiting failure. They are therefore unsuitable during the early life of a system and on very highly reliable systems since the failure sample is likely to be too small to permit meaningful analysis. This is one of the reasons that confidence building throughout development is so important.

One approach to predicting reliability before a system exists has relied on knowledge of its parts and their levels of reliability estimated from testing a large population of identical parts. The combining theory usually assumes that these parts fail independently, i.e. the presence of one does not modify the reliability of another. Unfortunately in complex items such as software, mutual independence is the exception rather than the rule. Even if the performance in service of two systems containing software is known, coupling them together with any degree of complex data interchange is likely to give significantly worse reliability than normal statistical combination would predict. Any structural calculation in the software case should take account of this lack of independence of failure, and achieved reliability should ultimately be assessed from observation of the total system functioning in its intended environment.

5.5 Process-based software reliability assessment

Assessment of software reliability based on examination of the planned and actual processes employed is an immature and imprecise discipline. None the less, it is vital that some view is formed during development of the reliability likely to be achieved by a given software-based system, and of whether this is likely to meet the requirements that have been established.

In the absence of specific methods and tools to perform this task, this document provides a structure within which the software reliability assessment issue can be addressed at early stages, before the software itself exists as a product. Inevitably, the structure implies the use of expert judgement (based on experience and historical data) concerning the achievability of reliability targets, the credibility of plans and in monitoring ongoing development processes. The underlying aim is to enable confidence in reliability to be built in parallel with the building of the software product.

5.6 Product models

5.6.1 General

Software product properties describe the state of a system at any point in time without regard as to how it was achieved. Examples of software product properties that can be expected to affect reliability include the following:

- a) code size;
- b) degree of conformance to accepted or predefined notions of good structure;
- c) type of software, such as that constrained to real time operation;
- d) language characteristics.

Data such as these are used by software product property models to estimate the level of reliability likely to be achieved by a given piece of software.

5.6.2 Use of fault data

Fault data in their simplest form provide a record of the numbers of faults identified in a software component by fault-finding activities (such as design reviews, Fagan inspections (see 3.16, note 3), or testing) during each of a number of phases of the software development cycle. Often this basic scheme is refined to include categorizations to show the severity of each fault and its origin (e.g. specification fault, design fault).

Whatever level of detail is chosen for fault recording, it is vital that the scheme is applied consistently across a range of projects. Only in this way can a valuable body of historical data be assembled, for use in predicting reliability.

The predictive method has two stages. First, past data should be analysed to identify patterns of possible predictive use. Commonly found patterns include a tendency for fault-prone components to continue to be so, and for levels of faults found at early stages to be correlated with later fault and failure levels; the strength of these relationships and their exact nature varies considerably between organizations, which is why data specific to the organization is so important. Stronger (and thus more useful for predictive purposes) relationships are usually found where there is a high degree of consistency in the processes which are used for software development.

The second stage of the predictive process is to apply predictive relationships derived in the first stage to the current project, so obtaining reliability (or other related) predictions, on the basis of early life fault data. The accuracy of these predictions depends on the current project being broadly comparable with those whose data were used to derive the relationships used, and can be expected to increase as more data from the current project accumulates, as development progresses. However, these methods can never be as accurate as those based on analysing specific failure data from the program in question (see 5.6.3); their value lies in having objective reliability-related information at a relatively early stage of development.

5.6.3 Failure data models

Failure data comprise either a sequence of the times at which failures have been observed, or (less precisely) counts of the numbers of failures observed in each of a consecutive sequence of time intervals. Data should be analysed statistically, to measure quantities such as failure rate at the current time, and to predict future levels by extrapolating forward any observed reliability growth achieved during testing, as a result of fault removal.

There are many statistical models for dealing with this problem, but no single model can be relied upon to give accurate predictions in every particular context; rather, a number of different models need to be used in each case, and their performance analysed to establish which should be given the greatest credence for that particular set of data.

The practical implications of applying these models are as follows.

- a) Testing should be carried out in a way which (at least approximately) satisfies the assumptions underlying the statistical models.
- b) Failure data should be collected from testing.
- c) Some expertise may be required to carry out the analyses.

5.7 Process models

5.7.1 General

Process-based assessment of software reliability aims to use information available at whatever stage of the development cycle has been reached, to establish confidence (or otherwise) in the reliability of the software being developed. Clearly, the amount of information available grows as development proceeds, and the nature of the information varies a great deal. Moreover, there is no standard method for approaching the task, which inevitably involves judgement to supplement and make use of the available data.

In the earliest stages of development, assessment involves establishing the feasibility or reliability targets set for the software, and the credibility of plans drawn up to achieve the target reliability. Later, assessment focuses on ensuring that the plan is being carried out in a way consistent with the target, and that intermediate products are of the requisite quality. Finally, failure data methods can be applied to measure reliability of the final product.

5.7.2 Assessing target reliability

The first possible assessment point is when a target has been set for the contribution which software is required to make to system reliability. At this point, the concern is to show that the target which has been set is achievable; it is better to discover at the outset that a target is unlikely to be achieved, than to spend a lot of money discovering this. Assessing target achievability is an application of the 'look before you leap' philosophy.

Targets should be expressed in quantitative terms, as for example a failure rate or a probability of failure on demand, or an integrity level.

Once a target has been set, its achievability should be assessed by analogy with existing systems, and the level of reliability that they have achieved. At moderate levels of reliability, it should be relatively easy to find systems which are sufficiently similar and sufficiently reliable to establish confidence in the achievability of the target. For very high integrity systems, this will often not be the case, especially if there are few similar software-based systems with which to compare. In these cases, careful judgement should be exercised to establish whether the available evidence generates sufficient confidence to proceed with the development. In all cases, a certain degree of judgement is unavoidable, since there will never be a perfect analogy for a new software-based system.

5.7.3 Assessing the planned process

Once a target has been established to be achievable, plans of various kinds can be produced to describe the process by which the software will be developed. These plans should now be assessed, to establish that the planned approach represents a credible way of achieving the target reliability. Planning a development which is believed to have a strong chance of success can make it easier later on to show that the target reliability has indeed been achieved.

The use of process models as the basis of project plans is recommended, as these not only show the component tasks, etc., but make very clear how they relate to each other.

There are no special techniques for establishing the credibility of a plan. In the ideal situation, data will be available to show how past projects using the same or sufficiently similar process models were carried out, and the achieved levels of reliability. Comparison of the plan with these historical records can be used to build confidence in the plan.

More realistically, either very little data will be available, or the data will not be from sufficiently similar projects. In these situations, a review of the plan by appropriate experts is the best substitute.

NOTE. An 'expert' does not necessarily have to be a 'highly paid external consultant'. More often than not a colleague who has worked on a similar project is suitable.

5.7.4 Assessing the actual process

During development, the process should be monitored to ensure its adequacy with respect to the reliability target. The aim should be to assure that plans are adhered to, and the quality of the work carried out. Inevitably, there will be some changes of plan during development; when this happens, assessment should focus on assuring that the changes do not adversely affect reliability.

The time to think about monitoring is when the project is being planned. This enables reviewing during the project to ensure that the plan is being adhered to, and that where divergences become necessary, they do not prejudice achievement of reliability targets.

Similarly, the definition of adequacy should be agreed beforehand, as should methods by which the quality of the work will be assessed during development.

It is impossible to define precise relationships between quality achievements during development, and reliability at the end of development. Nevertheless, intermediate targets should be set which are acknowledged to be consistent with the targeted level of reliability.

5.7.5 Other uses of process models

In addition to the assessment issues discussed above, which relate to the reliability of a specific system, process models may be used to help generate other types of information. These include the following:

- a) the relative merits of various software techniques relating to achieved reliability, e.g. the relative merits of particular languages for particular applications;
- b) the significance of different activities during the software life cycle in achieving reliability, e.g. it might be concluded that the coding activity has less significance than the specification activity.

5.8 Applicability and limitations of methods

5.8.1 General criteria

Methods of predicting and assessing software reliability fall into one of three main categories according to the type of measurements required and the times at which they are made (see 6.1).

- a) Assessment of development process (see 5.8.2 and 6.2).
- b) Assessment of product properties (see 5.8.3 and 6.3).
- c) Assessment of software failure (see 5.8.4 and 6.4).

Criteria for selecting methods in each category are described in 5.8.2 to 5.8.4 respectively.

Methods in the three categories complement one another and are not mutually exclusive. However the ultimate aim should be to measure the achieved reliability of the software before it is put into service. Selection of methods depends upon whether the assessment is being made by a developer or for purposes of procurement, the phase in the development life cycle which has been reached, the required level of system integrity, and the availability of the necessary data.

Methods which provide a quantitative assessment of reliability should be preferred to those that provide an index of merit or qualitative assessment, however the latter may still be useful, for example in helping to improve achieved reliability. It is important that any method chosen should be theoretically sound.

5.8.2 Process models

5.8.2.1 Criteria for selecting process models

Development process models (see 6.2) do not assess reliability as defined in this standard but may evaluate certain indicators which correlate with eventual software reliability. Even methods which provide no quantitative evaluation may be useful in improving achieved reliability by providing a qualitative assessment of factors which can be controlled by management during development and are known to affect reliability.

5.8.2.2 Regulatory requirements

For certain applications, particularly safety critical systems, development process assessment is mandatory. For example, the application of [5] to civil avionics software is required by the Joint Airworthiness Authority (JAA) regulations, and the use of the HSE Guidelines [6] may also be mandatory in some circumstances.

5.8.2.3 Recommendations for process model selection

During early development before a product is available, process models are the only type of assessment available. Process measurement is recommended as a means of quality management. Methods which yield quantitative predictions should be given preference.

Certain published approaches, e.g. [7] are highly flexible, require the setting of quantitative targets in the requirements definition phase and the assessment of the relevant attributes for comparison on completion of the project, and can incorporate other forms of reliability assessment, e.g. one based on failure data.

Inspections are recommended as a means of achieving reliable software and also as a means of quality control. Statistics of instances of non-conformance found by inspection may be used to identify software components which are likely to be problematic later on. Measures derived from these statistics, e.g. non-conformance density (usually referred to in the context of inspections as 'defect density'), might be found to be useful indicators of reliability, but predictions can be based on these indicators only by reference to an adequate database of previous projects, and such predictions should be treated with caution.

The recommendation here is that any process measurement (whether imposed by regulations or otherwise selected) is followed by a reliability assessment based on observation of the system in operation, and that the accuracy of predictions based on early indicators is evaluated from such observations.

5.8.3 Software properties models

5.8.3.1 Criteria for selecting properties models

Software properties models (see 6.3), like process models, do not assess reliability as defined in this standard but may evaluate indicators, which in the case of properties models are usually measures of internal static attributes.

Quantitative methods should be preferred.

Any measure chosen should be theoretically sound, i.e. the attribute which is being measured should be well-defined and measurement should be made on a meaningful scale (see 4.4).

The correlation of the indicator value with the achieved level of reliability should be well established with reference to earlier products whose operational reliability has been assessed.

5.8.3.2 Recommendations for properties model selection

Measurements of the 'structuredness' of software may be useful, and in any case require that configuration management is practiced, which is an essential prerequisite for both the achievement and assessment of operational reliability.

Methods which purport to yield measures of 'complexity' as a single number should be treated with caution, since recent work has shown that complexity is a composite attribute [3] and that some published measures have less predictive capability than a count of lines of code [8].

As in the case of process models, the accuracy of predictions based on any indicator derived from internal static product measures should be established by subsequent measurement of software reliability in operation.

5.8.4 Stochastic reliability models

5.8.4.1 Criteria for selecting stochastic reliability models

Stochastic reliability models provide indirect measures of reliability derived from direct measures of failure over operating time. The data from which these direct measures are extracted should be collected during system operation or during a realistic trial (see 6.4).

Such models can be classified into one of three main categories: general statistical techniques, black-box models or structural models (see 6.4.2).

There are many different software reliability models. The following criteria are recommended as a basis for evaluating a model for a particular application.

a) *Predictive accuracy* describes the capability of the model to predict future failure behaviour and should be determined by comparing failure rates and failure intervals predicted by the model with actual values observed.

b) *Usefulness* refers to the ability of the model to estimate quantities needed by managers and engineers in planning and managing software development projects. The degree of usefulness needs to be assessed from the importance of the measures provided.

c) *Quality of assumptions* implicit in the model should be checked by determining the degree to which it is supported by actual data. The clarity and explicitness of an assumption should be judged to determine whether a model applies to particular circumstances.

d) *Applicability* indicates the potential for use of the model across a range of different systems and different development environments. If a model gives outstanding results but for only a narrow range of systems or environments the model should not necessarily be discounted.

e) *Simplicity* of a model has three aspects. The most important is that it should be simple to collect the data that is required for the model. Secondly, the model should be simple in concept so that personnel without extensive mathematical backgrounds are able to understand the nature of the model and its assumptions. Finally a model should be easy to implement as a program that is a practical management and engineering tool.

5.8.4.2 Recommended general statistical techniques

The application of general statistical techniques (see 6.4.3) for preliminary data analysis is recommended but it should be noted that most such methods cannot give long-term predictions. Isotonic regression is a useful method of estimating instantaneous failure rate and is simple to apply. Graphical analysis of data is always advisable (see 6.4.5).

5.8.4.3 Recommendations for black-box models

5.8.4.3.1 Accuracy of black-box models

It is very important to appreciate that 'All models are wrong, but some are more wrong than others!' In other words, no model makes completely correct assumptions about the failure process, and all are therefore theoretically suspect. Some models have been found to give acceptably accurate results on some data sets, but none have been found to be consistently better than others over all data sets. Users are therefore advised not to trust a single model but to apply several and judge their predictive accuracy as described in 6.4.20.

It should be noted that adaptive modelling (see 6.4.20.5) may remove much of the bias from the estimates, and that a weighted combination of several estimates (see 6.4.20.6) may be more accurate than a single estimate.

It should also be emphasized that the successful application of failure data models depends crucially on adequate data collection and on the use of realistic operational profiles during trials. It should also be noted that little work has been done to validate long-term estimates of reliability growth and such estimates should therefore be treated with caution.

5.8.4.3.2 Usefulness of black-box models

Most black-box models potentially provide useful estimates such as expected time to next failure, failure rate, expected number of faults that will be found, etc.

'Input domain' models (see 6.4.19.2) estimate probability of failure per input. They may be useful given a well-defined operational profile, and where 'probability of failure per demand' is a quantity of interest.

Some models in the 'miscellaneous' sub-category (see 6.4.19) are limited to estimating the 'number of faults in the product' and cannot be used directly to predict reliability. An example is fault 'seeding' which also has the disadvantage that it depends upon a sample of faults which is likely to be atypical of the population.

Availability models (see 6.4.19.4) are useful where system requirements include a certain maximum length of down-time, but where recovery from software failure is concerned, down-time should be measured by time to restore service, not time to repair.

5.8.4.3.3 Quality of assumptions of black-box models

Black-box software reliability growth models based on incorrect assumptions, such as a uniform fault activation rate, e.g. Jelinski-Moranda (see 6.4.12.2), should be treated with caution. This applies to many of the early published models.

Models which allow for imperfect and delayed debugging are preferable but careful data extraction may still permit useful estimates to be obtained from models which assume perfect and immediate fault removal.

Most models assume that operating conditions are constant, and will give totally inaccurate estimates if this is not the case. The various 'environmental factors models' (see 6.4.16) provide a promising but relatively untried approach to overcoming this limitation.

5.8.4.3.4 *Applicability of black-box models*

Black-box models can be applied to almost any type of software, provided that adequate data are collected and that the operating conditions are as assumed by the model (i.e. in most cases, they are assumed constant and representative of the service environment).

An important exception is the assessment of very high reliability, for which stochastic reliability models are not suited since they depend upon analysis of a reasonably large sample of failure data, whereas the observation of any failure of a system with a very high reliability requirement means that it is inadequately reliable (see 6.5).

5.8.4.3.5 *Simplicity of black-box models*

Given the availability of a software tool to carry out the numerical search to estimate the parameters from the failure data, the use of black-box models is fairly straightforward, but the initial design and construction of such tools is a major task.

5.8.4.4 *Recommendations for structural models*

Structural models (see 6.4.21) are recommended in cases where software components whose reliability can be estimated from previous operational experience are being re-used or where parts of a system are being modified and it is necessary to estimate the reliability of the new version, taking into account the operational reliability of the unchanged parts together with the reliability of the new or modified components as measured during trial. However, the use of such models requires measurement of the extent to which each component is exercised relative to the total system operating time. This is not always easy, and may require instrumentation of the software.

5.9 Procedures

5.9.1 *General*

A variety of procedures are needed to support and enable software reliability assessment. A summary of the important management implications follows; details about procedures are contained in clause 7.

5.9.2 *Data collection procedures*

The data collection to be carried out should be derived from specific assessment objectives established for each project, and wherever possible in accordance with existing practices and procedures.

Procedures are needed to ensure the proper management of this potentially wide-ranging data collection and storage activity, some of which may be properly regarded as a project activity, but some of which may be co-ordinated. It is preferable that the overall control of data collection is co-ordinated to ensure consistency of approach and quality management of data, but much of the data will necessarily be provided by project resources; this should be allowed for when determining project budgets and time-scales.

Data required will typically include:

- a) failure and fault data, including times and circumstances of failure and details of actual or potential consequences;
- b) process data, such as methods and techniques used and resources employed;
- c) product data, i.e. information about size, structure, languages used and the version which failed, for example.

Data will have many sources, including personnel employed at various stages of development, existing management systems and system users (i.e. those who fill in failure reports).

5.9.3 *Project management procedures*

Software reliability assessment is an activity which should occur throughout system development and use from the earliest stages. The only exceptions to this are where conscious decisions are taken not to carry out any reliability assessment activity. Even in these cases, a management framework is needed to ensure that the decisions are taken in a timely and proper fashion.

Project management criteria and procedures should address software reliability assessment, indicating the need for assessment activities at all stages, as outlined in 5.1 and providing organization-specific guidance on how the various actions needed should be documented and approved. The guidance provided should not be prescriptive, as selected techniques should reflect the nature of particular systems and the uses to which they will be put; also, the range of techniques available will change as the subject area matures. However, the guidance given should cover product-based assessment as described in 5.4 and process-based assessment as outlined in 5.5.

Emphasis should be given to the need for evidence to support any claims for reliability, whether they are in response to stated requirements, or are made for other management, marketing or licensing purposes. It is equally important to stress that the activity should start very early in the process so that reliability targets can be set and reviewed, and corresponding procedures put in place to ensure that any necessary data are collected in a timely fashion.

5.9.4 *Maintenance considerations*

After release, systems will be subject to maintenance activities during their operational life. With software, it is not uncommon for the maintenance processes employed to differ from those employed during initial development. The potential for relaxation of discipline and rigour, from the high levels present during initial development, represents a significant threat to the benefits gained from adopting a rigorous approach in the first instance. With the accompanying scope for structural deterioration within the software design as a consequence of less

disciplined maintenance this may impact upon the continuing achievement of the systems target reliability. In particular, where an external certification requirement may exist, e.g. for safety, the effects of a relaxed maintenance environment may transcend the issues of reliability.

It is important to recognize that the maintenance of software requires elements of the initial development process if it is to be effective. Therefore, adoption of the same degree of rigour in maintaining software as in its initial development would be highly desirable. Where such rigour cannot be adopted, for whatever reasons, all changes to the maintenance process should be recorded in detail, in order that management can observe their effects and, if necessary, ensure that any corrective action required is taken.

6 Software reliability assessment techniques

6.1 Classification of techniques

Reliability is an external dynamic attribute of a product and is measured indirectly (see 4.4.4.2). Methods of assessing software reliability can be divided into the following categories, based on the other types of measure from which they derive the measures of reliability.

- a) Software development process models use measures of intermediate products (e.g. specification and design documents) generated during software development, or measures of the development process itself.
- b) Software property models use internal measures of static attributes (i.e. properties) of the source code of the delivered software.
- c) Stochastic reliability models use statistical analysis of past failure data, or of information about the levels of reliability of individual models.

Internal measures and process measures that are correlated with the reliability of the delivered software are referred to as indicators of reliability. Most property and process models predict reliability from the values of certain indicators measured for the particular product in question on the basis of the correlation observed between past values of the indicators and past levels of reliability for a sample of previous software developments. (Some do not predict reliability as defined in this standard but predict other quantities such as 'defect density' or 'number of faults in the product'.)

To validate such a model, i.e. to establish how accurately the indicators predict reliability, it is necessary to measure the degree of correlation. This requires the measurement of the reliability of each software product in the sample, which in turn requires the application of stochastic reliability models to the records of behaviour in trial and operation of those sample products.

Stochastic reliability models base their estimates on the observed behaviour of the product in question, and so provide reliability measures more directly than process or property models, whose estimates of reliability are more indirect and are derived from imperfect correlation.

A few models combine features of more than one class, e.g. the Musa calendar time model (see 6.4.13.6) combines a stochastic reliability growth model with measures of the development process, and structural models (see 6.4.21) combine elements of stochastic reliability models with measures of software structure.

6.2 Software development process models

6.2.1 Introduction to process models

6.2.1.1 Characteristics of process models

Software development process models attempt to derive measures of the quality of the delivered software product from measures of the development process. Most do not provide measurements of reliability as defined in 4.5. At best they are capable of estimating the values of indicators whose correlation with operational reliability should be established separately.

There are two main approaches that depend on process measurement.

- a) Inspection statistics (see 6.2.2) are collected during formal inspection of intermediate development products and can be used to estimate defect density which is a possible reliability indicator.
- b) Qualitative assessment of good practice (see 6.2.3) analyses documentary evidence that certain techniques which may be expected to yield high quality software have been applied during development.

6.2.1.2 Data required by process models

Process models require process and resource measurement. These may include measurements of elapsed time and effort expended on particular development activities and counts of defects detected in the intermediate or end products. From these the values of certain indicators can be estimated.

In order to correlate the indicator measurements with operational reliability, these models additionally require measurements of the same indicators together with measurements of operational reliability for a sample of previous similar software products.

6.2.1.3 Estimates provided by process models

Not all process models deliver estimates of reliability indicators. Most provide only measurements of static quality attributes, or simply extra confidence that the developers have done a good job.

6.2.1.4 *Advantages of process models*

Given that adequate 'corporate memory' exists to validate such models as predictors of reliability, process models are in theory capable of providing predictions early in the software development life cycle. They can indicate if it is feasible to meet a given reliability target and provide warning of potential 'hot-spots' within the system (e.g. particular modules which are likely to cause problems) so that management attention can be focused on these with the aim of avoiding future problems due to unreliable system components.

Where very high reliability is required in software, stochastic reliability models are not capable of providing the necessary degree of confidence, and process assessment is often the best that can be achieved, at the expense of having no quantitative estimate of software reliability. This is discussed in more detail in 6.5.

6.2.1.5 *Disadvantages of process models*

Process models only estimate software reliability very indirectly, if at all. Any estimate based on a process model should always be followed by observation of the delivered system in trial and operation.

6.2.2 *Inspection statistics*

6.2.2.1 *Introduction to inspection statistics*

Formal inspections as originally described by Fagan [9] have been found to be an effective method of achieving reliable software, and are in routine use by many developers. Inspections are carried out at the end of each phase of the development life cycle. The intermediate output products from the phase are read by at least two inspectors (neither of whom is the author of the material under inspection) during a carefully organized series of inspection meetings carried out by a structured team with well-defined roles. The objective of each meeting is to compare the output products of the phase with the input products from which they were developed and note any instance of non-conformance. (These are commonly referred to as 'defects' by those who practice inspections and this term will be used here in discussing inspections, although it is generally deprecated.) For example the detailed design is inspected against the high-level design, and the source code is inspected against the detailed design. The author of the material then re-works it to remove any defects and the re-work is signed off by the 'moderator' (who heads the inspection team) before the next phase of development may begin. If too many defects are found, the material should undergo re-inspection after re-work. By this means a reasonable fraction of defects are removed at each phase so that only a small percentage survive to become faults in the delivered product.

The method requires the careful collection of statistics of defects found in each inspection, the size of the material inspected and the effort expended in inspection and re-work. Statistical norms are established by the organization, and an anomalous count of defects may mean that management action is required. For example a low count might indicate an inadequate inspection, or a high count might indicate a difficult module.

Although measures of product quality are derived from these statistics, the method depends on observation of the development process, and so is classed as a software development process model rather than as any other type.

6.2.2.2 *Data required for inspection statistics*

6.2.2.2.1 The following data are collected for each inspection.

a) Count of defects broken down as follows.

1) Severity

- Major: could have led to system failure in operation.
- Minor: could not have led to system failure in operation, e.g. a misleading comment in source code.

2) Category

- Missing: something specified at the higher level was omitted from the material under inspection.
- Wrong: something specified at the higher level was implemented in the material under inspection, but incorrectly.
- Extra: something was included in the material under inspection which was not specified at the higher level.

3) Type

- A code for the type of non-conformance as defined in the inspectors' detailed checklist.

b) Size of material inspected. Usually measured as thousands of lines of source code (KLOC). (Sizes of other intermediate products, e.g. high-level and detailed design specifications, are usually converted into the number of KLOC they are expected to generate after coding.)

c) Size of material re-worked. Also measured in KLOC, broken down into material added, modified and deleted.

a) Effort. Usually measured in person-hours, broken down into preparation, inspection and re-work time.

6.2.2.2.2 In order to use inspection statistics to predict the behaviour in service of a product currently being developed, the total data required are as follows:

- a) the data detailed in **6.2.2.2.1a**) to d) for inspections in the current development project;
- b) the data detailed in **6.2.2.2.1a**) to d) for all the inspections during the developments of a sample of previous similar products;
- c) records of the numbers of faults activated during the entire service life for each previous similar product in the sample; and
- d) total amount of execution time during its entire service life for each previous similar product in the sample.

6.2.2.3 *Estimates provided by inspection statistics*

The following measures are commonly derived from inspection data.

- a) Inspection and re-work rates: (size of material)/(effort).
- b) Defects detected: counts of defects found and removed by re-work. These may be broken down by inspection and by module, and further broken down by type, category and severity.
- c) Detected defect density: (count of defects detected)/(size of material).
- d) Defects remaining: estimated counts of defects still remaining after inspection. These may be broken down as in b).
- e) Density of defects remaining: (Estimated count defects remaining)/(Size of material).
- f) Defect removal efficiency: The percentage of defects originally present which are detected in a given inspection and removed by re-work.

Inspection and re-work rates (see item a)) measured for previous inspections are useful for planning future inspections. Their derivation is straightforward and poses no conceptual difficulties. Defects detected (see item b)) and their densities (see item c)) as measured for the current project and compared with previous projects (or among various modules within the current project) are useful indicators of the following problems:

- inspections which may be inadequate (unexpectedly low count);
- modules which may be problematic (unexpectedly high count);
- types of mistakes to which development personnel are prone, or to which the methods of development are conducive (unexpectedly high proportions of certain types of defect).

These measures can be used to improve the skill of personnel (by learning to avoid common mistakes), the effectiveness of development (by selection of methods which are not conducive to common mistakes) and the efficiency of inspections (by including on the inspectors' check-lists the more frequently found types of defect).

NOTE. It is axiomatic in Fagan inspection that the statistics should not be used to judge the abilities of personnel, due to the effect of the rigour of inspections and inherent difficulty of the material produced on the statistics.

Estimates of defects remaining (see item d)) and their densities (see item e)) are often used as measures of software quality. It is extremely important to note that such estimates are not measures of reliability as defined in this standard. Their usefulness is reviewed in detail in **6.2.2.5**.

It has been claimed [10] that inspection statistics can be used to predict the number of faults that will be activated during the service life of the product. Such predictions are based on estimates of defect removal efficiency and counts of faults detected in service for previous similar products. The validity of such predictions is also discussed in **6.2.2.5**.

It is possible that the careful extrapolation of trends in non-conformance counts may yield useful predictions of reliability in service, and examples of attempts to do this have been published [11].

6.2.2.4 *Advantages of inspection statistics*

Inspections have been found to be cost-effective in achieving high quality software. Case studies have shown that software modules that have been inspected tend to be more reliable than those which have not [12], and that the average effort in man-hours required to detect defects during inspection is less than one fifth of that required to detect a fault during testing [12]. Given a sufficiently comprehensive corporate memory covering a period of many years during which inspections have been used consistently and including records of experience of the resulting products in service, it may be possible to correlate trends in density defects detected with operational reliability.

6.2.2.5 *Disadvantages of inspection statistics*

Estimates of defects remaining and their densities (**6.2.2.3d**) and e)) are at best static measures of software source code attributes which may be indicators of software reliability. Even considered as such, they are very indirect measurements and should be derived from correlation with data from previous similar projects. Estimates of defect removal efficiency (see **6.2.2.3f**)) are also of dubious value since they depend upon estimates of **6.2.2.3d**) and e) from earlier projects.

In particular 6.2.2.3d) is often referred to in the literature on inspections as the 'number of defects delivered', and 6.2.2.3e) as the 'error rate' or 'defect rate'. This usage of terms is seriously misleading and should be avoided for the following reasons:

- a) they are static measures and not rates over time;
- b) many defects are in the 'minor' category and by definition cannot cause failure in operation, so are not faults (or 'defects') in the delivered product; and
- c) defects which are easily detected in inspection are not necessarily similar to faults which are easily activated in operation.

Predictions of the number of delivered faults using simplistic methods require very strong assumptions. For example it is assumed in [10] that there are only two inspection stages and that these both have the same defect removal efficiency. This has been observed to be false [13]. It is also assumed that previous products now at the end of their operational life are similar to the product currently in development and that all of their defects have been discovered by the ends of their operational lives. Such methods should be regarded with the deepest suspicion.

The 'number of delivered defects' sometimes estimated using inspection statistics is similar to the 'total number of faults in the product' which some of the early stochastic reliability models purport to estimate. Such estimates (of total number in a population based on observation of a small sample) pose notoriously difficult statistical problems [14] and in the context of software reliability it is doubtful if they are meaningful measures of anything at all.

6.2.3 Qualitative assessment of good practice

6.2.3.1 Introduction to assessment of good practice

Several standards and guidelines adopt an approach which is based on a qualitative assessment of good development practice. Examples are the European Organization for Civil Aviation Electronics guidelines [5] which apply to airborne avionics software, and the Health and Safety Executive guidelines [6] which apply to process control systems.

Such guidelines usually recommend certain practices which should be adopted in order to deliver software of a given level of integrity, depending on the criticality of its application, i.e. the likely severity of the consequences of a failure in the worst case. For example, [5] defines five levels of integrity of software depending on whether it is executed in a system whose failure may have 'catastrophic', 'hazardous', 'major', 'minor' or 'negligible' consequences for the flight of the aircraft.

In many cases (e.g. [5]) the guidelines are used as a basis for certification by a regulatory authority (e.g. Joint Airworthiness Authority) that the system (including the software) is adequately reliable.

6.2.3.2 Data required for assessment of good practice

Such guidelines often provide checklists for recording the presence or absence of desirable practices, or define documents that are to be provided by the manufacturer to the certifying agency as evidence of having followed the guidelines. This evidence may include specifications, test plans, test reports and a 'statement of achievement'.

6.2.3.3 Estimates provided by assessment of good practice

A review of the documentary evidence may confirm that an adequate development process has been carried out. Essentially the assessment is bureaucratic. A given document either does or does not exist and if it exists has been read and assessed independently to be of adequate quality. This in turn is evidence that a given development function has or has not been performed adequately.

Predictions of operational reliability can be deduced only by appeal to previous successful service of similarly assessed software products.

6.2.3.4 Advantages of assessment of good practice

Assessment of good practice can take place from the earliest stages of software design and development and provides information in advance of operation. In cases where a high integrity is required assessment of good practice provides one important method of gaining confidence in the software but it should not be used on its own.

6.2.3.5 Disadvantage of assessment of good practice

Such guidelines are usually concerned with the achievement of high integrity software rather than its assessment, and provide no quantitative assessment of operational reliability with respect to software failure. In many cases such a quantitative assessment is deliberately not required. For example, although the higher level guidelines [15] which invoke [5] recommend that (in most cases) the system level reliability is estimated quantitatively, they specifically preclude the incorporation of probability of software failure in the calculation of system reliability.

6.2.4 Formal methods

The set of techniques known as formal methods is a means of verification that is applied mainly to software which is required to exhibit very high reliability. The software requirements specification is written in a mathematical language. The specification includes a precise definition of conditions to which its internal state is to conform at each stage of execution, referred to as preconditions (which should be satisfied on entry to any module or code segment) and post-conditions (which should be satisfied on exit). A proof of correctness is then constructed to demonstrate that when the code is executed, each post-condition is a logical consequence of the preceding precondition. This proves formally that the code is a correct implementation of its specification.

The fact that formal methods have been used is often cited as a demonstration that the software is certain to be free of certain classes of faults.

6.3 Software property models

6.3.1 Introduction to software property models

6.3.1.1 Characteristics of software property models

Software property models evaluate static measures of the software product. These methods often form part of a more general quality assurance activity, intended to assess quality measures other than reliability such as maintainability, portability, etc. and may be classified as follows:

- a) software science is a well-known approach proposed by Halstead [16] which can derive reliability indicators from source code;
- b) complexity measures represent how difficult a piece of source code is to understand, and have been used as reliability indicators.

Quality factors for software include 'reliability' but defined as a combination of more basic static properties.

6.3.1.2 Data required by software property models

These usually consist of direct measures of the source code and may be evaluated by analysis of the code using a compiler or static analyser.

6.3.1.3 Estimates provided by software property models

These include estimates of such quantities as 'number of faults in the delivered source code' and numerical measures of 'complexity'.

6.3.1.4 Advantages of software property models

Static analysis of source code may provide valuable information in advance of operation, and may reveal potential faults that are unlikely to be activated in testing. The measures derived may be indicators of operational reliability.

6.3.1.5 Disadvantages of software property models

The measures derived are not measures of reliability as defined in this standard but are at best only indicators of reliability. As with process models, the predictive usefulness of these indicators should be established separately, which requires the application of stochastic reliability models to failure data obtained in operation, and correlation over a sample of products.

Most of the models apply solely to imperative programming languages.

Many published software property models have been found to have little value for predicting reliability in practice.

6.3.2 Software science

6.3.2.1 Introduction to software science

Halstead [16] proposed an approach named 'software science', in which quantitative relationships between various direct measures collected from the source code are hypothesized.

6.3.2.2 Data required by software science

The number of distinct operators used and the number of distinct operands used are added to give the 'vocabulary size'. The total number of operators used and total number of operands used are added to give the 'program length'.

The 'program volume' is a measure of the total amount of information contained in the source code, and is given by the length multiplied by the logarithm of the vocabulary size. Many other direct measures are used.

6.3.2.3 Estimates provided by software science

The 'number of faults in the delivered code' is then estimated as '(program volume)/3000', since 3000 is assumed to be the 'number of elementary discriminations between errors'.

Many other quantities can be estimated, but are not relevant to reliability.

6.3.2.4 Advantages of software science

It is capable of providing estimates before testing, and the necessary direct measures are easy to evaluate using a static analysis tool. It is intuitively appealing that the number of mistakes a programmer is likely to make should depend upon the amount of information contained in the code.

6.3.2.5 Disadvantages of software science

The theoretical foundations of 'software science' are now generally regarded as being unsound, and the apparent experimental support for the theory initially reported is thought to be due to poor experimental design. Hamer and Frewin [17] have criticized the theory on these grounds, and also point out that the prediction is solely of coding faults, not of all possible types of fault.

6.3.3 Complexity measures

6.3.3.1 Introduction to complexity measures

Models of this type all attempt to formalize the common sense expectation that the larger and more complicated software is, the more likely it is to contain faults. Lipow and Thayer [18] and Akiyama [19] have applied regression to establish quantitative relationships between such measures as code size and the number of branches in a program, to the number of faults delivered. The McCabe complexity metric [20] is also claimed to correlate with the number of faults in a piece of software.

6.3.3.2 Data required by complexity measures

Complexity measures are usually derived from the analysis of source code. Most approaches depend upon the graphical representation of the program structure. A graph consists of a set of points or nodes connected by line segments or edges. In a directed graph each edge is assigned a direction indicated by an arrow on the diagram, and the number of edges which arrive at or leave a given node are referred to respectively as the in-degree and out-degree of that node. A control-flow graph or flowgraph is a directed graph which includes a start node which has in-degree zero and a stop node which has out-degree zero.

If each node represents a software instruction, then any execution of the software is represented by a walk through the flowgraph beginning at the start node and ending at a stop node. Any node with out-degree greater than one represents a decision point in the code. For any flowgraph other than the simplest, there are many possible walks or paths through it.

6.3.3.3 Estimates provided by complexity measures

McCabe's complexity measure [20] is defined by the equation:

$$V_g = e - n + 2$$

where:

- V_g is the complexity measure;
- e is the number of edges;
- n is the number of nodes.

It measures the number of linearly independent walks through the graph. McCabe recommended that no software module should have a complexity number greater than 10.

Fenton [3] discusses and compares several approaches to measuring 'complexity' (including McCabe). A method is recommended which also uses the flowgraph approach but in a less simplistic way which depends on decomposing the flowgraph into prime subgraphs (i.e. those which cannot be decomposed further). It is then possible to ask more penetrating questions about the nature of control flow complexity, but it is pointed out that 'complexity' is not a simple attribute and cannot be measured as a single scalar number.

A number of testing tools use the flowgraph representation to guide testing and in particular to measure test coverage. This may be measured at various levels. For example, the test effectiveness ratio (TER) may be calculated in the following ways.

- a) Proportion of all statements executed (i.e. nodes visited).
- b) Proportion of all edges visited.
- c) Proportion of all possible paths executed.

6.3.3.4 Advantages of complexity measures

Theoretically, complexity measures can indicate which modules are too complex and will therefore exhibit a high density of faults. These may then be redesigned or decomposed into simpler modules.

The test effectiveness ratios provide a measure of the thoroughness with which a given set of test data exercises the code.

6.3.3.5 Disadvantages of complexity measures

Simple metrics such as McCabe's are not meaningful measures of what is generally understood as total 'complexity'. The complexity of a piece of software might reside in its data structure rather than its control flow. A simple example is given in [3] of a program designed in two different ways to perform the same required functions as follows.

- a) Using a simple data structure and a fairly complex flow of control gives a McCabe number of 7.
- b) Using a complex data structure and a 'straight through' flow of control, the McCabe number is 1.

It is obvious that the McCabe metric does not capture all aspects of what is intuitively understood by the attribute 'complexity' and that this is in fact a composite characteristic which needs to be decomposed into several more well-defined attributes in order that these can be measured.

In one case study [21] the actual 'difficulty' of each module in a sample was measured by the number of times it had to be re-worked before release. It was found that the correlation between that measure and the McCabe number of the module was slightly worse than its correlation with the count of lines of source code in the module. (The same study reported a similar finding for the Halstead 'volume' metric). It appears that such metrics provide little more information than 'lines of code'.

A high level of test coverage is obviously desirable, but attempts to predict the reliability of the delivered system from the achieved value of the TER measures have proved to be ineffective. TER measures are therefore poor candidates as reliability indicators.

6.3.4 Quality factors

6.3.4.1 Introduction to quality factors

Another approach which has been widely used and copied is that of Walters and McCall [22]. This depends upon the definition of a number of quality factors, as originally proposed by Boehm et al. [23], and McCall et al. [24].

6.3.4.2 Data required by quality factors

Each factor depends upon a number of quality criteria (denoted primitive characteristics by Boehm et al. [23]). Reliability depends upon completeness, accuracy and consistency (Boehm et al. [23]), or consistency, accuracy, error tolerance and simplicity (McCall et al. [24]). The criteria depend in turn upon a number of measures which are