BRITISH STANDARD

BS 5760 :
Part 8 : 1998

Reliability of systems, equipment and components

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

ICS 21.020; 35.080

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

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

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 It supersedes DD 198 : 1991, which is withdrawn.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.

blank

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 [1].

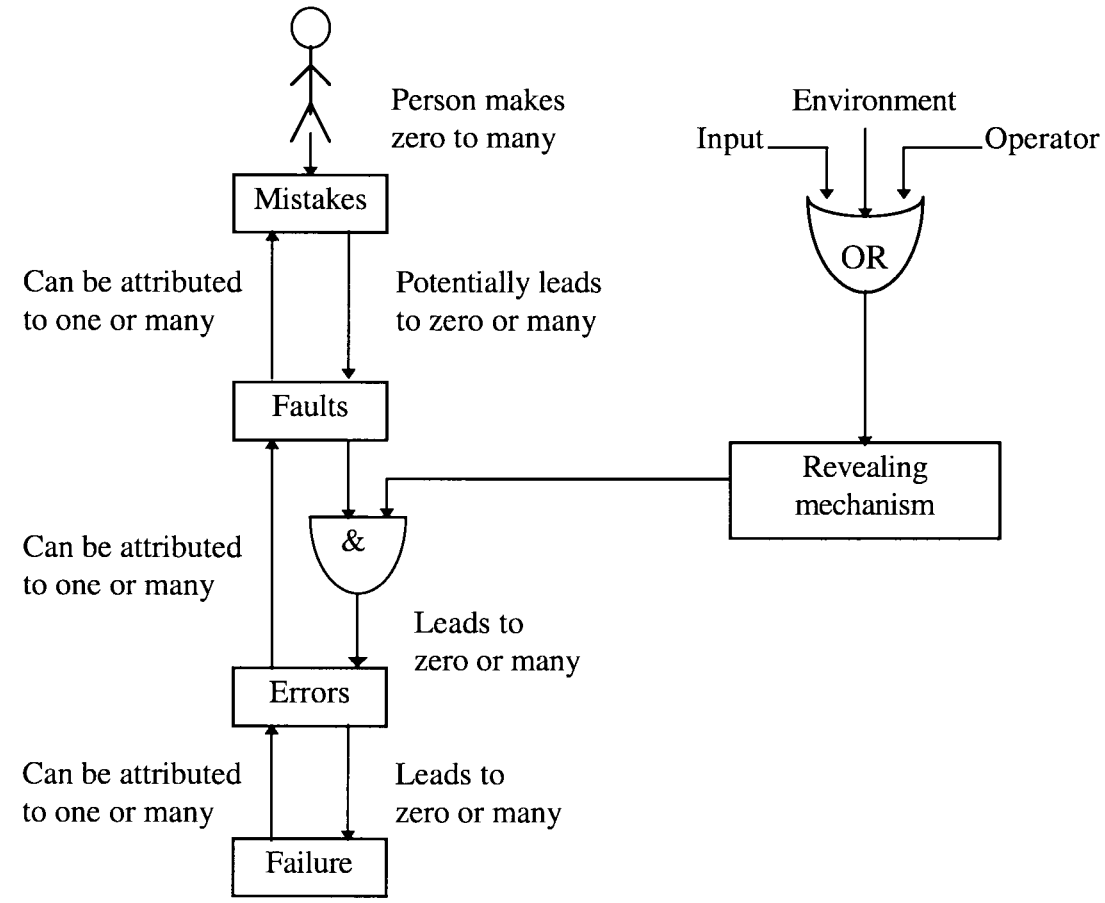


Figure 1. Mistake, fault, error, failure relationship

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.

a) 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.

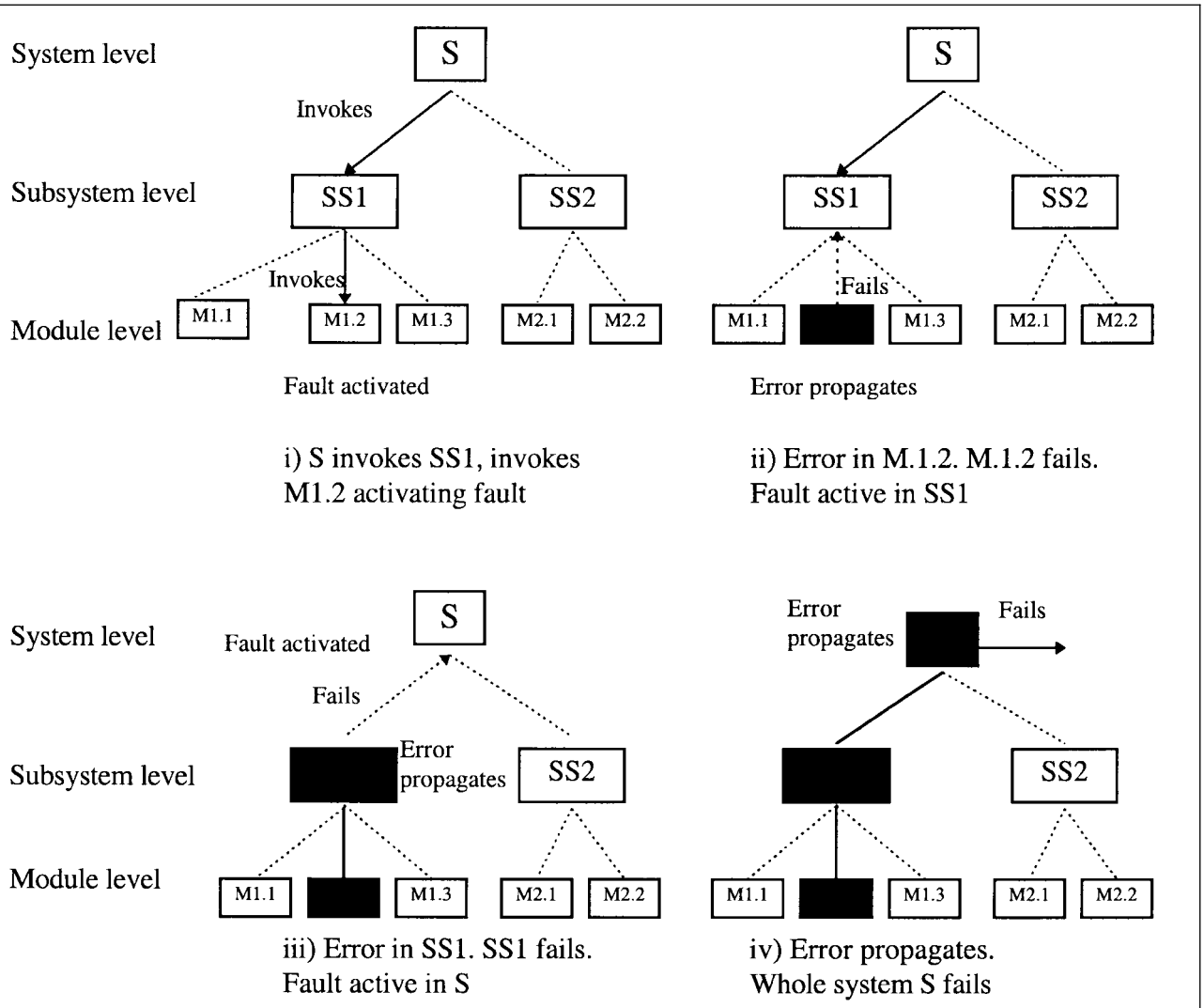


Figure 2. Software failure mechanism in a simple hierarchical system

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 **6.2.2.2.1**a) to d) for inspections in the current development project;

b) the data detailed in **6.2.2.2.1**a) 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 (**6.2.2.3**d) 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 **6.2.2.3**f)) are also of dubious value since they depend upon estimates of **6.2.2.3**d) and e) from earlier projects.

In particular **6.2.2.3**d) is often referred to in the literature on inspections as the 'number of defects delivered', and **6.2.2.3**e) 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 directly observable.

For example, Walters and McCall [22] measure simplicity by counts of product components which are independent of all others and which have single entry and exit points. They define functions to combine the basic measures in order to derive measures for the criteria and these in turn are combined to give a rating for each quality factor.

6.3.4.3 *Estimates provided by quality factors*

Boehm et al. [23] define the following quality factors: portability, reliability, efficiency, human engineering, testability, understandability and modifiability.

McCall et al. [24] suggest the following: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability and interoperability.

The end result is an index of merit which is intended to be a measure of the overall quality of the software.

6.3.4.4 *Advantages of quality factors*

Like complexity metrics, quality factors theoretically purport to be static product measures which are indicators of operational reliability. They could therefore be used prior to system trial or operation to predict the behaviour of the system. They might also enable the project manager to 'trade-off' one factor (e.g. reliability) against another (e.g. efficiency) in the detailed design and coding phase if development resources were limited.

6.3.4.5 *Disadvantages of quality factors*

The quality factors are often not defined in a measurable way, so that validation of the postulated correlation between these metrics and a measure of reliability is difficult, if not impossible. No strong relationship between the values of these metrics (or an 'index of merit' derived from them) and operational reliability has been established. Kitchenham [8] has criticized this approach both for this reason and because it provides no clear indication of what the relationships are between different factors that might be 'traded-off' against each other.

6.3.5 ***Fault tolerance***

Fault tolerant design of software is a means of protecting the system from the effects of activation of residual latent faults. The software is designed to include several independently written versions of each module. The outputs of all versions are compared and the value upon which a majority of the versions agree is taken to be the 'correct' overall output (referred to as *N version programming*). An alternative approach is to submit the outputs of the different versions to an *adjudicator* which checks that essential integrity constraints have not been violated. Versions are invoked in turn until the output of one of them satisfies the adjudicator (referred to as the *recovery block scheme*). The intention is that the error resulting from the activation of a fault within one version will be detected and corrected by reference to the outputs of other versions so that it does not propagate and result in a system failure.

Evidence of the use of fault tolerance is often cited as a reason for having confidence that the software will exhibit high reliability.

6.4 Stochastic reliability models

6.4.1 *Introduction*

6.4.1.1 *Nature of stochastic reliability models*

Stochastic reliability models represent the mechanism of system failure in a probabilistic (or 'stochastic') way and have been used for some time to estimate the levels of reliability of hardware systems with respect to both physical failure and design failure (see **4.2**). They can also be used to estimate system reliability with respect to software failure, and many of the models can be applied both to hardware design failure and to software failure.

Stochastic reliability models which apply to software can be classified hierarchically (see **6.4.2**). The characteristics of each class, sub-class and individual model, the data that is needed in order to apply them, the quantities that they estimate, and the advantages and disadvantages of each, are described in **6.4.3** to **6.4.5**. Methods for comparing the accuracy of the estimates obtained from different models, for recalibrating estimates to improve their accuracy, and for combining different estimates into a single quantity, are described in **6.4.6**.

Technical detail has been kept to a minimum in **6.4**, but a few mathematical terms are unavoidable. More detailed mathematical descriptions of some important classes of model and of individual models are contained in annex B and in the referenced sources, e.g. [25]. The statistical methods used to assess the accuracy of estimates, and to recalibrate and combine them, are contained in annex C.

6.4.1.2 *Data required by stochastic reliability models*

All stochastic models derive their estimates of reliability from data which should be collected while the system is being used in a way which is as similar as possible to that in which it will be used in service. In practice, this means during a realistic trial, or during actual service. There are a few exceptions, e.g. seeding and tagging models (see **6.4.19.3**).

General statistical techniques (see **6.4.3**) and black-box parametric models (see **6.4.9**) derive their estimates from the failure history of the system. The following data are required.

a) Records of system operating time and/or software execution time. It is extremely important that the measure of time used is meaningful, i.e. that it is a true measure of the extent to which the software has been exercised and any latent faults in it exposed to the possibility of activation. Calendar time is rarely adequate.

b) Records of all failures, including both calendar time at which they occurred and how their occurrences are distributed over execution time. Type of failure (symptoms observed), severity (seriousness of the consequences) and location (which installation was affected) may also be required.

c) For each failure, the identity of the latent fault which was activated. There is a crucial distinction between the first activation of a hitherto undetected fault, and a failure due to a subsequent activation of a fault which is already known. Data should be collected in such a way that these two types of event can be separated.

d) The identity of the product which failed and its baseline version number.

e) Information about the operating environment and the way in which the product was being used when it failed. In order to predict software reliability in service on the basis of observations of failure during test or trial, random testing should be used during the test or trial. This means that the input cases should be selected according to the probability distribution which governs the relative frequencies with which they will be encountered in service. A definition of such a distribution is referred to as an operational profile. Random testing does not mean selection of input cases with equal probability.

Again there are a few exceptions, e.g. input domain models (see **6.4.19.2**) analyse the distribution of failure over the set of possible inputs, instead of over execution time.

Availability models (see **6.4.19.4**) also require records of system recovery times and/or of times to diagnose and correct faults. Other particular models have specific additional data requirements.

Structural models (see **6.4.21**) require the following data.

– The reliability of each individual software module, and of each interface.

– Information about the execution time spent in executing each individual software module. For most models, a measure of the execution profile, i.e. the proportion of total execution time spent within each module, will suffice.

6.4.1.3 *Estimates provided by stochastic reliability models*

Stochastic models estimate values of various reliability measures for the system. The following are some examples.

a) Probability that the software will not fail in a given period of operation.

b) Mean time to next activation of a new software fault.

c) Current failure rate.

d) Predicted failure rate after a given further period of trial and fault correction.

e) Expected number of faults that will be activated in a given further period of operation.

f) Further time required for trial and fault correction in order to achieve a defined target value for any of a) to e).

Not all classes of model are capable of estimating all of these quantities.

6.4.1.4 *Advantages of stochastic reliability models*

Stochastic models estimate reliability measures for the particular system being studied. They are dynamic, i.e. they estimate measures of the behaviour of the product in service.

Although all reliability measures are indirect (see **4.4**) stochastic models are less indirect than other techniques, and do not depend upon data from previous similar projects, nor upon correlation with indicators (see **6.1**).

Statistical tests are available (see **6.4.20**) to assess objectively the accuracy of the estimates, and techniques are available to improve their accuracy by recalibration and combination of different estimates.

Stochastic models are the only means of estimating the reliability of the product in operation in order to validate the more indirect predictions obtained from process models (see **6.2**) and software property models (see **6.3**).

6.4.1.5 *Disadvantages of stochastic reliability models*

Since stochastic models analyse data collected while the system is being used in a realistic way, they can be applied only very late in the development life-cycle, when the system is complete (or nearly so). They cannot be used to predict system reliability in the early phases of development, e.g. requirements specification and high level design.

Although stochastic models provide the most direct estimate of reliability measures, these estimates have often been found in practice to be inaccurate. In particular, a bias towards optimism has been observed. (However, note the methods for assessing and improving accuracy described in **6.4.20.1**.)

It is crucial that the failure history is observed under realistic conditions. The reliability of a system containing software is notoriously sensitive to changes in operating conditions. In practice it is not easy and may not be possible to ensure that a trial is conducted under conditions which are representative of the operating environment that will be found in service. If the conditions are not representative the reliability estimates derived cannot be expected to be accurate and may be wildly misleading. Furthermore it is often precisely the unexpected or unusual conditions which give rise to software failures in service. (Environmental factors models (see **6.4.16**) which specifically predict the effect of environment, are an exception.)

6.4.2 *Classification of stochastic reliability models*

6.4.2.1 *Basis of classification*

Stochastic reliability models can be classified as shown in figure 3 according to the assumptions made regarding the mechanism of failure and the ways in which the process of failure is modelled mathematically.

At the top level, stochastic models can be classified into the following major categories.

a) General statistical techniques (see **6.4.3**) are general purpose methods of statistical analysis, and can be applied to many types of data, not solely to failure histories. They make no assumptions about the mechanism of failure, and so do not really qualify as 'models'.

b) Black-box parametric models (see **6.4.9**) disregard the internal structure of the system and represent only its externally observable failure behaviour. They model the mechanism of failure using formulae which incorporate parameters, i.e. unknown quantities which are estimated from the failure data.

c) Structural models (see **6.4.21**) mathematically represent the internal interactions between components of the system and combine their individual levels of reliability to estimate total system reliability. Since structural models treat the components as black-boxes, they are dependent on black-box models.

The classes of model within each of these major categories are very briefly described in **6.4.2.2** to **6.4.2.4**.

6.4.2.2 *Classification of general statistical techniques*

Most of these methods of analysing failure data fall into one of the following classes.

a) Graphical analysis is used to picture the failure data and reveal important features (see **6.4.5**).

b) Exploratory data analysis (EDA) applies a battery of standard statistical tests to search for patterns in the data (see **6.4.6**).

c) Monotonic regression fits the most appropriate completely monotone function to the failure rate (see **6.4.7**).

d) Time series analysis is a set of standard statistical techniques for studying stochastic processes (see **6.4.8**).

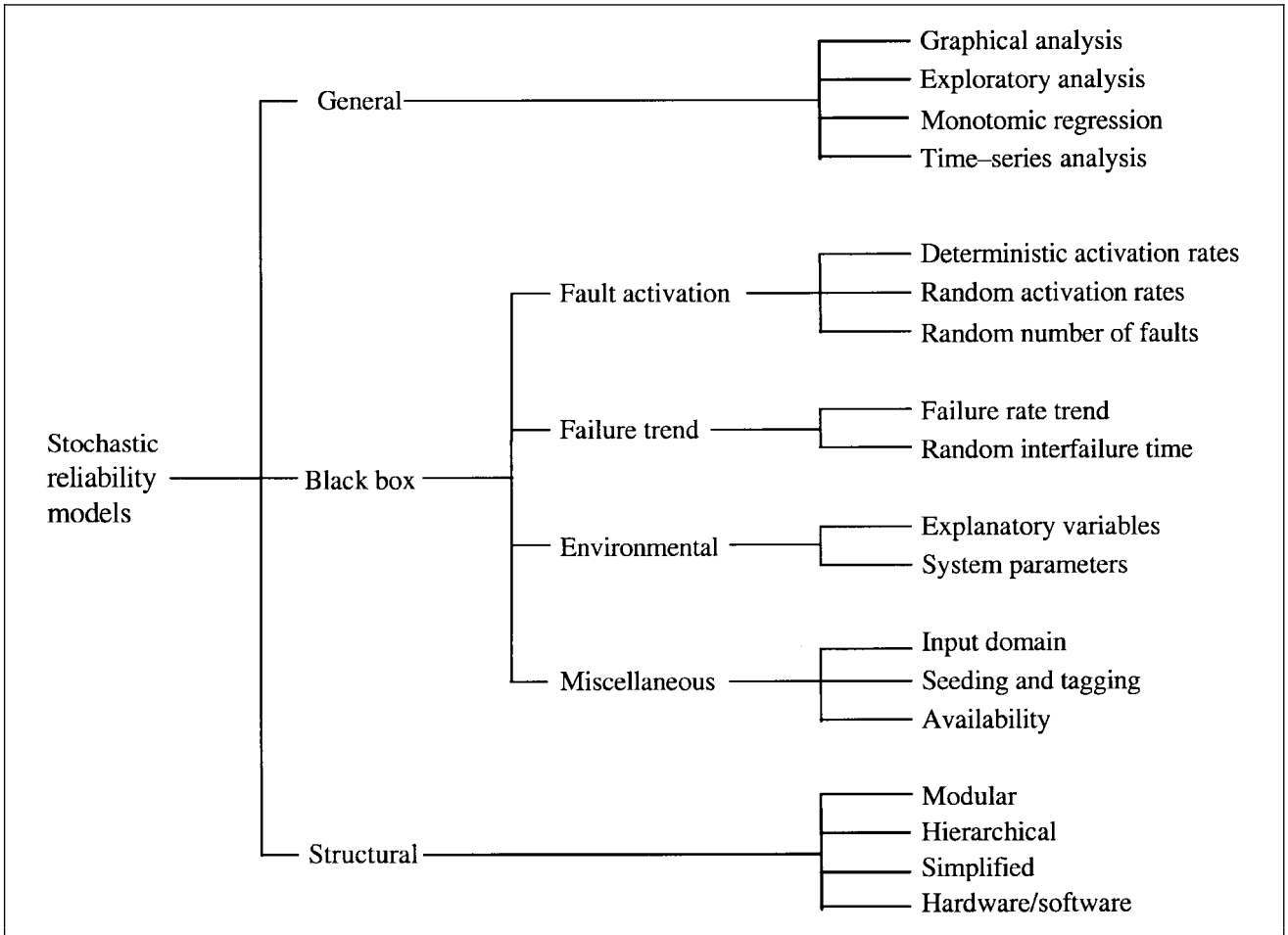


Figure 3. Classification of stochastic reliability models

Graphical methods, EDA and a simple type of monotonic regression are useful for preliminary analysis of failure data, as described in **6.4.4**.

6.4.2.3 *Classification of black-box parametric models*

Most black-box models fall into one of the following classes.

a) Fault activation models (see **6.4.11**) represent failures as being due to the activation of latent design faults within the system, each fault being activated at random with a certain rate. There are three main types as follows.

1) Deterministic models assume that the activation rates are all identical or are determined by a given mathematical function.

2) Random activation rates models assume that the activation rates themselves are random variables.

3) Random number of faults models assume that the number of faults in the system is a random variable.

b) Failure trend models (see **6.4.14**) are not concerned with fault activation, but model the failure rate or interfailure times directly. The two main types are as follows.

1) Failure rate trend models fit an exponential function to the trend and the failure rate of the system (see **6.4.15**).

2) Random interfailure time models represent times to system failure as random variables (see **6.4.15.3**).

c) Environmental factors models (see **6.4.16**) take account of the influence of variations in operating conditions on system reliability. The two main types are as follows.

1) Explanatory variables models estimate a base failure rate for the system, and adjust this using 'explanatory variables' to take account of the extent to which various environmental factors that might affect the failure rate are present (see **6.4.17**).

2) System parameters models correlate the values of certain measures of the hardware/software system with the software failure rate (see **6.4.18**).

d) Miscellaneous models (see **6.4.19**) comprise a few approaches which do not fit readily into the categories above, including some which do not estimate 'reliability' as defined in this standard. The three types described in this standard are as follows.

1) Input domain based models estimate reliability with respect to the space of possible inputs, from the proportion of a sample of inputs which cause the system to fail, instead of analysing the occurrence of failure over operating time (see **6.4.19.2**)

2) Seeding and tagging methods estimate the number of faults in the system from the number found during testing, by applying statistical methods originally used to estimate size of animal populations from the sizes of samples of tagged animals recaptured (see **6.4.19.3**).

3) Availability models incorporate the time to restore service or time to diagnose and correct faults so as to estimate availability as well as reliability (see **6.4.19.4**)

Fault activation models and failure trend models are collectively referred to as stochastic reliability growth models since they take account of the improvement in system reliability as software faults are activated and corrected.

6.4.2.4 *Classification of structural models*

Structural models combine the levels of reliability of individual components to produce an estimate of the reliability of the whole system (see **6.4.21**). Most such models which apply to software fall into one of the following four classes:

a) modular software models represent system behaviour as a random transfer of control between a set of modules with certain average lengths of time for each visit and certain probabilities of failure of each module;

b) hierarchical models represent system behaviour as the invocation of lower modules in a hierarchical structure by modules in the level immediately above;

c) simplified structural models make no detailed assumptions about the transfer of control between modules, but simply assume that each module occupies a given proportion of system operating time;

d) hardware/software models represent the behaviour of a mixed hardware/software system, with each hardware or software component sharing a certain proportion of total system operating time.

6.4.3 ***General statistical techniques***

6.4.3.1 *Characteristics of general statistical techniques*

These are standard techniques used for statistical analysis of many types of data and are not restricted solely to failure data. They are used to investigate general features of data such as trend over time, clustering of data points, evidence of non-randomness such as cyclic correlation, and the presence of 'outliers' or other anomalies.

6.4.3.2 *Data required by general statistical techniques*

The general statistical techniques described here are of use on the same type of data that is used by black-box parametric models, i.e. history of failure over execution time. The data sets in question typically exhibit reliability growth as faults are removed from the system, i.e. a graph of the accumulated number of faults activated against execution time will show a decreasing slope, indicating a decreasing failure rate.

Such failure data may be in either of the following two forms.

a) time to failure: a record of the execution time between each activation of a new fault; or

b) failure count: the number of new faults activated, and the amount of execution time, in each of several successive periods.

These are illustrated in figures 4 and 5 respectively.

It is extremely important to note that reliability growth is generally observed only with respect to the activation of new faults. This is illustrated by the example of a data set collected from a real software product shown in figure 6.

Unless repeated activations of faults already detected are removed from the data before analysis, the main trend may be obscured. A history of all failures collected from a product which is executed under operating conditions which remain constant over time, and which is not undergoing fault correction, would be expected to show a constant failure rate.

6.4.3.3 *Estimates provided by general statistical techniques*

The different techniques provide many different types of estimate. Examples are:

a) estimates of instantaneous failure rate at given points in time or of average failure rate over given time intervals;

b) measures of trend which indicate if reliability growth or decay is present, if the failure rate is constant, or if there are short periods of locally increasing failure rate. Most software reliability growth models assume the presence of reliability growth and will yield meaningless estimates in other cases. A preliminary evaluation of a trend may therefore be used to decide whether software reliability growth models should be applied (see **6.4.4**);

c) detection of clustering or periodicity in the data which may indicate that the operating conditions are changing or that other factors are influencing system behaviour;

d) detection of outliers: data points which differ greatly from the majority, for example an interfailure time which is several orders of magnitude greater than all the other interfailure times observed. The causes of these may require investigation.

6.4.3.4 *Advantages of general statistical techniques*

The general techniques described here can investigate features of the data which are ignored by black-box parametric models. Their lack of assumptions means that they are useful in detecting anomalies in the data which may violate the assumptions of parametric models and so invalidate results of applying them.

Graphical analysis, EDA and isotonic regression are simple to apply, particularly given the ready availability of statistical and graphical software tools on personal computers.

6.4.3.5 *Disadvantages of general statistical techniques*

The lack of modelling of the mechanism of failure means that these techniques are not suitable for making long term predictions of future reliability growth.

Although they are capable of revealing anomalies in failure data, they give no indication as to their cause, which should be established by independent investigation.

6.4.4 ***Preliminary analyses***

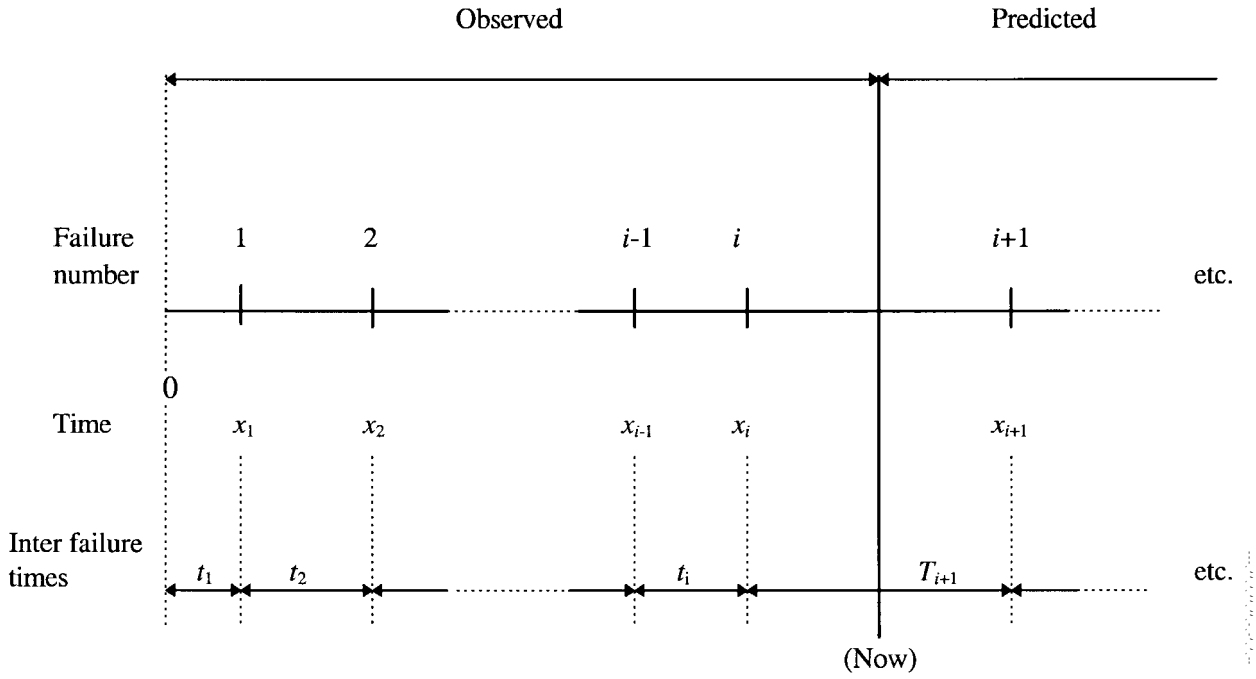
It is advisable to use several general techniques to perform a preliminary analysis before applying black-box parametric models, since these may reveal features of the data set which would render the parametric models inapplicable. Simple methods of analysis which do not require sophisticated statistical techniques including the following:

a) Graphical presentation (see **6.4.5**): data such as accumulated faults found and empirical failure rate are plotted against accumulated operating time. Important features such as obvious trend can often be detected by eye.

b) Exploratory data analysis (see **6.4.6**): straightforward statistical tests are applied to search for features of the data, such as trend, cyclic correlation, clustering, or periods of increasing failure rate.

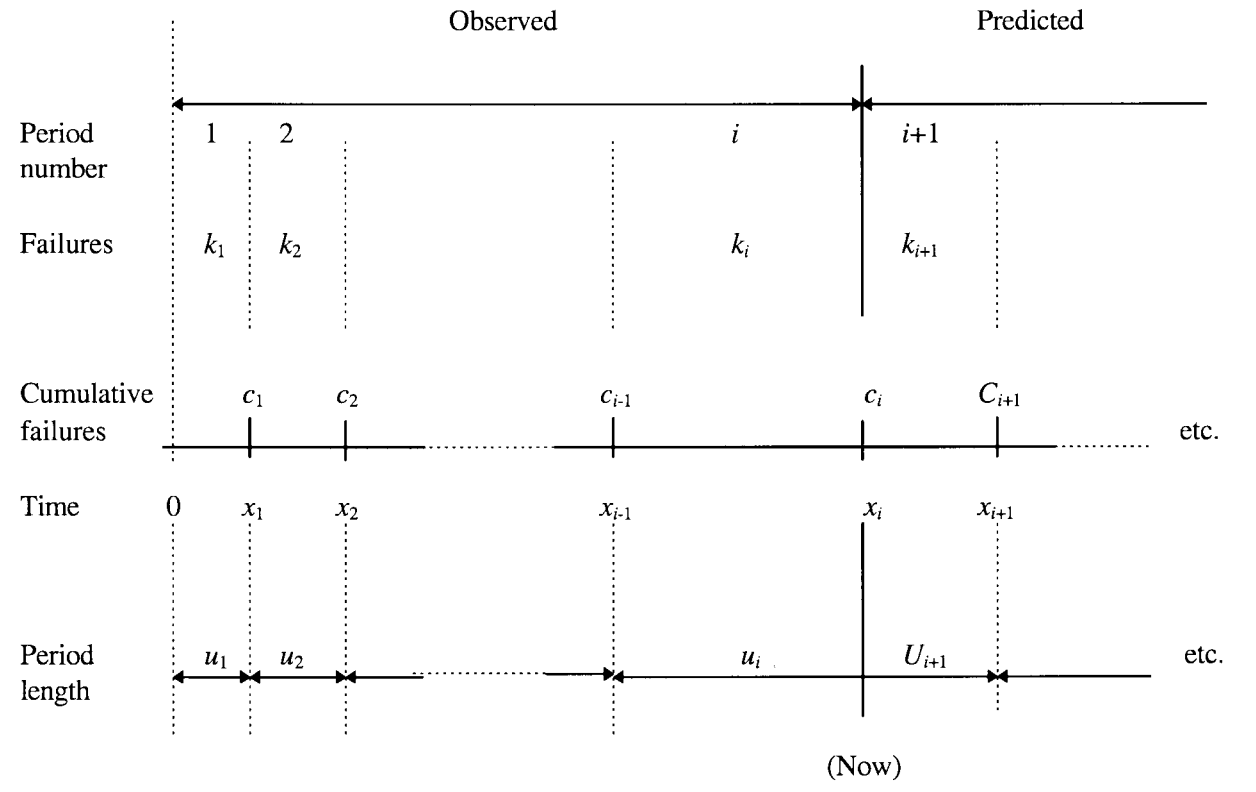
c) Isotonic regression: This is a straightforward graphical technique for estimating failure rate where reliability growth is evident. Some faults are more likely to be activated than others, so that the faults observed will tend to be ordered by their individual activation rates. The term isotonic indicates that this ordering is taken into account. Graphically, isotonic regression can be performed by drawing the smallest envelope of straight line segments that lies above the graph of accumulated faults found plotted against operating time. This envelope is referred to as the least concave majorant (LCM), and its slope provides an estimate of the failure rate at that point in time. Isotonic regression is a special case of monotonic regression (see **6.4.7**).

Figure 6 is an example of a graphical representation of a real data set showing the LCM and also the expected accumulated number of faults activated and the expected failure rate as estimated by one of the black-box parametric models described in **6.4.9**. The data set is of the 'failure count' variety. The numbered points are those at which the LCM touches the graph of the actual data, at which the LCM slope changes.



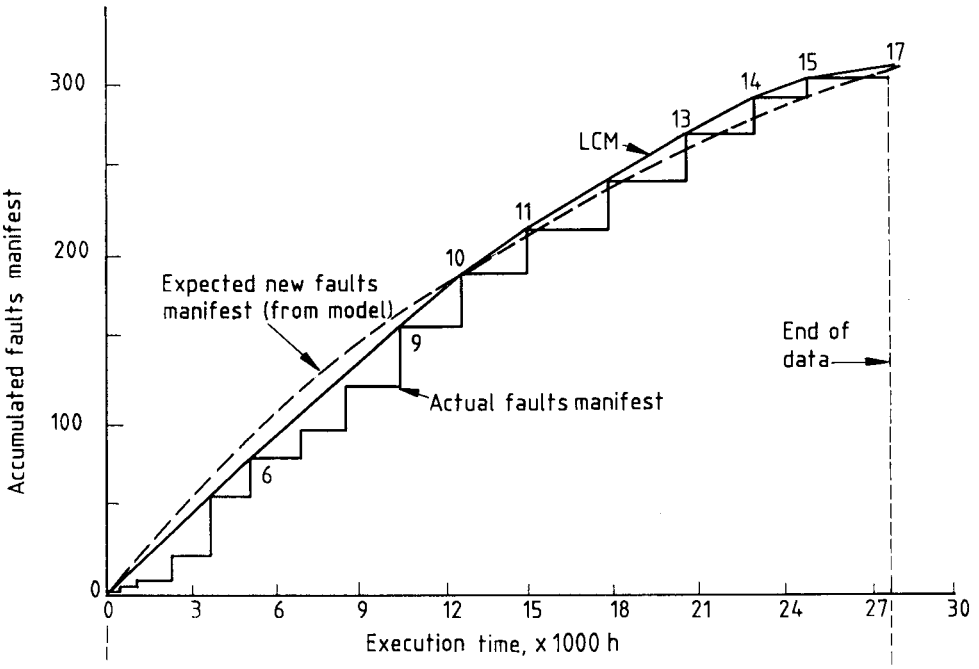
NOTE. $x_1 = t_1$, $x_2 = t_1 + t_2$, etc.

Figure 4. Fundamental reliability assessment problem: time to failure

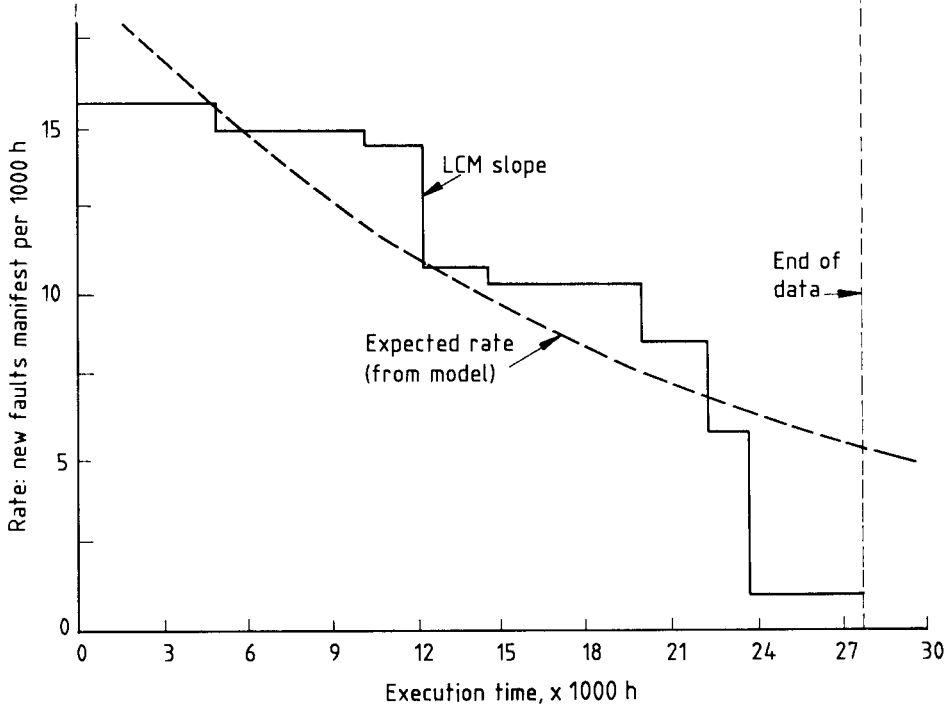


NOTE. $c_1 = k_1$, $c_2 = k_1 + k_2$, etc.

Figure 5. Fundamental reliability assessment problem: failure count data



(a) Accumulated first manifestations of faults against execution time



(b) Failure rate (1) Estimated from slope of LCM
(2) Derived from model

Figure 6. Example of failure history graphs and use of LCM

6.4.5 ***Graphical analysis***

The following are examples of useful graphs that may be plotted:

a) accumulated number $c(x)$ of faults activated plotted against accumulated execution time, x;

b) the logarithm of the accumulated number $c(x)$ of faults activated plotted against the logarithm of the accumulated execution time, x;

c) for failure count data, the empirical failure rate estimated by k_i/u_i, where k_i is the number of faults activated and u_i is the execution time in the ith period, for successive periods i = 1, 2, 3..., etc., plotted against accumulated execution time, x;

d) for time to failure data, the empirical failure rate in successive intervals of equal execution time u, estimated by $[c(x_i) - c(x_{i-1})]/u$, where x_i is the accumulated execution time up to the start of the ith interval, plotted against accumulated execution time, x, or against failure number, i;

e) for time to failure data, the ith times between failure (TBF), t_i, plotted against accumulated execution time, x, or against failure number, i.

6.4.6 ***Exploratory data analysis (EDA)***

EDA employs many techniques, including the graphical methods listed in **6.4.5** and numerical statistical tests. It can reveal features of the data such as the following:

– trend, e.g. by using the Laplace test, or by fitting a regression line to the graph of log $[c(x)]$ against log (x) to investigate how well a function of the form $c(x) = ax^b$ fits the data ($b < 1$ indicates reliability growth and $b > 1$ indicates increasing failure rate);

– serial correlation among times between failures, e.g., by plotting t_i against t_{i-1} (or against t_{i-2}, t_{i-3}, etc.) to see if the result is a random scatter plot;

– cyclic failure behaviour;

– clustering of failures;

– marked periods of increasing failure rate.

Anomalies in the data may be due to problems with the conduct of the trial, with the definition of measures, or with data collection procedures. The following are a few common examples:

– changes in the operating conditions are often revealed by increases in failure rate;

– imperfect fault correction (either not completely removing a fault, or introduction of new faults during modification) may lead to a cluster of faults being activated;

– inadequate measures of execution time may result in cyclic patterns of failure, e.g. if failures are recorded over calendar time the occurrence of weekends may be apparent;

– problems with data collection. Incomplete recording of execution time may show itself by an apparent increase in failure rate. The presence of several failures due to activations of the same fault before removal may appear as clustering.

If such anomalies are observed, the cause should be investigated. It may be necessary to review the way in which the trial is conducted, or to filter the data before further analysis.

6.4.7 ***Monotonic regression***

This method of non-parametric analysis involves fitting the best completely monotone function to the failure data by minimizing the sum of the mean square deviations between the function and the results of applying backward difference operators to the empirical failure rate over a succession of intervals [26]. It has been successfully applied to real data [27]. Isotonic regression (see **6.4.4**) is a special case of monotonic regression.

6.4.8 ***Time-series analysis***

Time-series analysis comprises a set of techniques for the statistical study of series of events, usually occurring over time, although other measures may be used in place of the time dimension. Their application is quite general and they are the subject of many publications. No assumptions are made about the nature of the process which generates the events. A general description of the techniques is given in Box and Jenkins [28].

The application of time-series analysis to software reliability growth data has been suggested by Dale and Harris [29] and Dale et al. [30]. Singpurwalla has attempted this [31].

Soyer [32] described a Fourier series model which is useful for investigating clustering of failures, and a random coefficient autoregressive model which is used to handle reliability growth or decay. Both are based on time-series analysis. Soyer concludes that the techniques constitute a potentially powerful tool, and that they can be used to predict future software failure.

6.4.9 ***Black-box parametric models***

These models treat the system as a 'black-box' whose internal structure and interactions between components are unknown, and represent only its external behaviour. They are classified into fault activation (see **6.4.11**), failure trend (see **6.4.14**), environmental factors (see **6.4.16**) and miscellaneous (see **6.4.19**) models as described in **6.4.2**.

Fault activation and failure trend models deal with the growth in software reliability as faults are removed and are referred to as software reliability growth (SRG) models. Similar models have been used to estimate hardware reliability growth during design fault removal and many SRG models apply equally well to hardware or software.

The fundamental problem for any SRG model is to estimate the future pattern of failure from data of past failures observed during a period of realistic operation, using conceptual models of the mechanism of failure and of the fault correction process.

An SRG model is a method of prediction which comprises three elements as follows.

a) A probabilistic model which specifies the probability distribution of future times to failure or of counts of failures activated in future periods of operation of a given length. These distributions are expressed as mathematical formulae which include parameters.

b) A statistical inference procedure to estimate the values of the parameters from past recorded failure data.

c) A prediction procedure which combines the mathematical formulae and the estimated values of the parameters to make quantitative estimates about the future probability of failure of the system.

SRG models are mainly classified on the basis of their probabilistic model as described in **6.4.2.3**a) and b). Their inference procedures and prediction procedures are sufficiently similar to merit a common description (see **6.4.10**).

However another important criterion of classification is the way in which the model represents the correction of faults after activation, sometimes referred to [33] as the repair model. This cuts across the classification illustrated in figure 3 and includes the following possible assumptions.

1) Effect of fault correction. This can be any of the following.

i) Perfect correction. All corrective actions are assumed to be completely successful.

ii) Imperfect correction. A corrective action removes a fault only with a certain probability.

iii) Fault introduction. Mistakes may be made during fault correction leading to the introduction of new faults.

2) Timing of fault correction. This may be as follows.

i) Immediate. Each fault is assumed to be corrected immediately after its first activation.

ii) Delayed. A fault may be activated repeatedly several times before being corrected.

Failure trend models can handle imperfect or delayed fault correction or fault introduction without needing to make any explicit assumptions. Fault activation models have to represent delayed and imperfect fault correction explicitly, and become more complicated by doing so [34]. If failure data is collected carefully and filtered so that only first activations of faults are input to the stochastic reliability model, it is usually possible to use a 'perfect immediate correction' model with accurate results even though the actual corrections are imperfect and delayed. Most of the models described in **6.4.11** to **6.4.14** assume 'perfect immediate correction', but can be extended to model imperfect and delayed fault correction.

6.4.10 *Estimation of stochastic reliability growth model parameters*

6.4.10.1 *Data required by stochastic reliability growth models*

SRG models require the same data as described for general statistical analysis (see **6.4.3** and **6.4.4**).

The data may be in either 'time to failure' or 'failure count' format (see figures 4 and 5), except that some failure trend models require 'time to failure' data.

Most SRG models require records of the first activation of each fault, since they assume 'perfect and immediate' fault correction. The time measure should be system operating time or software execution time. It is generally assumed that the failure data is collected during a period of trial or operation in which the operating conditions are the same as those which will be obtained during the period for which predictions are being made, and this may require random testing (see **6.4.1.2**).

6.4.10.2 *Inference procedure*

During the trial or operation of the system SRG models are applied repeatedly and the parameters are continually re-estimated as the data set of failures observed over execution time grows. Therefore there is no fixed value of the parameters from which reliability predictions are derived, since their estimated values change. In fact the instability of these estimates is a problem with many of these models.

The statistical inference procedure referred to in **6.4.9**b) is a method of estimating the value of the parameters. The methods commonly used are the following.

a) Optimization of an objective function as follows.

1) Maximum likelihood estimation (MLE); the objective function is an expression which represents the likelihood that the data will have been observed given that the assumptions of the model are 'true' and that the parameters have a particular value.

2) Least square distance (LSD); the sum of the squares of the differences of the observed data point from the corresponding values of a function derived from the model, such as $m(x)$ the expected number of faults activated by total execution time x, or $r(x)$ the failure rate after total execution time x.

3) The optimum value of the objective function (maximum for MLE, minimum for LSD) has to be found over all possible parameter values. No analytical solution is generally available i.e. there is no straightforward mathematical formula from which the optimum values can be calculated. It is therefore necessary to do a numerical search. Since most models have several parameters, this search is multi-dimensional and the procedure poses problems of its own.

b) Bayesian inference; a prior probability distribution function (PDF) for the values of the parameters is transformed into a posterior PDF that is conditional on the observed data, using Bayes' theorem (see annex B for the mathematical details). The calculations involve multidimensional integration of complex expressions so this method tends to be computationally intensive and has been limited in practice to a few of the simpler models.

c) Model specific methods; methods which can be used only for particular models, e.g. the estimation of the 'failure intensity per failure' parameter in the Musa-Okumoto logarithmic Poisson model (see **6.4.13.7**). Okumoto [35] demonstrates how this can be estimated by linear regression of failure intensity (estimated from the data) against the accumulated number of faults activated.

Various procedures are available for performing numerical searches, e.g. the Nelder-Mead simplex search [36], and an adaptation of Newton-Raphson iteration [37].

Several problems can arise in optimizing an objective function.

1) An optimum value of the objective function may not exist so that the search does not terminate. This problem has been studied for some particular models [38]. These cases need to be detected and the lack of solution reported by the software package used to perform the search.

2) The search may settle on a local maximum and not yield the best overall value.

The search procedure is independent of whether time to failure data or failure count data is in use. Model parameters can be estimated from either type of data, provided a slight adaptation is made to the objective function.

6.4.10.3 *Estimates provided by SRG models*

Once the values of the parameters have been inferred, they can be substituted into various formulae derived from the model in order to calculate numerical measures of future failure behaviour. (Where Bayesian inference is being used, a Bayesian procedure for estimating these measures is desirable. See annex B.)

Examples of quantities that can be estimated are listed in **6.4.1.3**. The following points should be noted.

a) Although the inference procedure is based on first activation only, the prediction procedure in most models can easily be made to estimate the occurrence of repeated failures due to activations of the same fault.

b) Whether time to failure or failure count data is available (see figures 4 and 5) affects only the inference procedure. Once the values of the parameters have been estimated, the prediction procedure can estimate either future times to failure or number of failures in a given future period of operation with equal ease (for nearly all models).

6.4.10.4 *Advantages of SRG models*

SRG models are specifically designed to cope with reliability growth, which is normally observed in software failure data. In theory SRG models are capable of predicting long term behaviour of the product under the same conditions of operation and fault correction. For example they could be used to predict the cost of corrective maintenance of a software product over its entire life.

Statistical confidence intervals can be placed around the estimates. Provided a large set of failure data is available on which to base the estimates, a high degree of confidence can be placed in them. SRG models are therefore very useful for predicting the behaviour of ordinary commercial software where a modest level of reliability is acceptable.

6.4.10.5 *Disadvantages of SRG models*

In practice the estimates may not be accurate, and if several different models are applied they may lead to different estimated values for the same measure. Therefore the accuracy of the estimates should be assessed and they should be recalibrated if necessary and estimates from different models may be combined as described in **6.4.20**.

Little confidence can be placed in estimates which are based on a very small set of data, and great caution should be exercised when attempting to make long term predictions from a short period of observation. This has particular implications for the assessment of very high levels of reliability. If high integrity software such as is used in safety-critical applications is under scrutiny, then either it will exhibit no (or very few) failures, in which case only a low confidence is justified that its reliability is sufficiently high, or else it will exhibit many failures, in which case there is a high degree of confidence that it is inadequately reliable. SRG models are therefore not suitable for the assessment of high integrity software.

6.4.11 ***Fault activation models***

Fault manifestation models regard software failure as due to the activation of latent faults. Models in this class make the following assumptions.

a) The software contains a set of latent faults.

b) Each fault is activated independently of all the others.

c) Each fault is activated at its own particular rate.

d) Failures due to the activation of a single fault occur at random, and on average are homogeneous over operating time.

e) On activation, a fault is immediately and perfectly removed from the system.

Assumption d) means that time to activation of a single fault is a random variable, and successive activations constitute a random process which is stable over time (on average, the frequency of activation does not change). Assumption e) means that the times to failure of the whole system are the times of first activation of each fault. This is illustrated in figure 7.

The faults shown in figure 7 are activated at different rates. Faults with higher rates of activation tend to give rise to system failure before those with lower rates, and are detected and corrected earlier in the life of the system. However since faults are activated at random, this is true only on average, e.g. fault 5 is detected before fault 2, fault 3 and fault 4 although it is activated less frequently overall. In mathematical terms, the time to activation of an individual fault is said to be an exponentially distributed random variable, and the times to failure of the whole system are said to be order statistics from a set of independent random variables. Models in this class are therefore known as exponential order statistic (EOS) models (see annex B).

Assumption c) means that the faults in a system are characterized by a set of activation rates. Each different sub-class of model is distinguished by the way in which the distribution of these rates over the faults is modelled. Individual models are distinguished by their assumptions about the precise distribution, and about the number of faults in the system.

Most models in this class include a parameter which represents the number (or the expected number) of faults in the system, and either one or two other parameters which represent the distribution of activation rates.

6.4.12 *Deterministic activation rate models*

6.4.12.1 *Introduction*

In a deterministic model the activation rates of each individual fault are assumed to be identical or to be specified by a formula. The reliability of the whole system increases as faults are activated and corrected. Most deterministic models assume that faults are corrected perfectly and immediately upon activation.

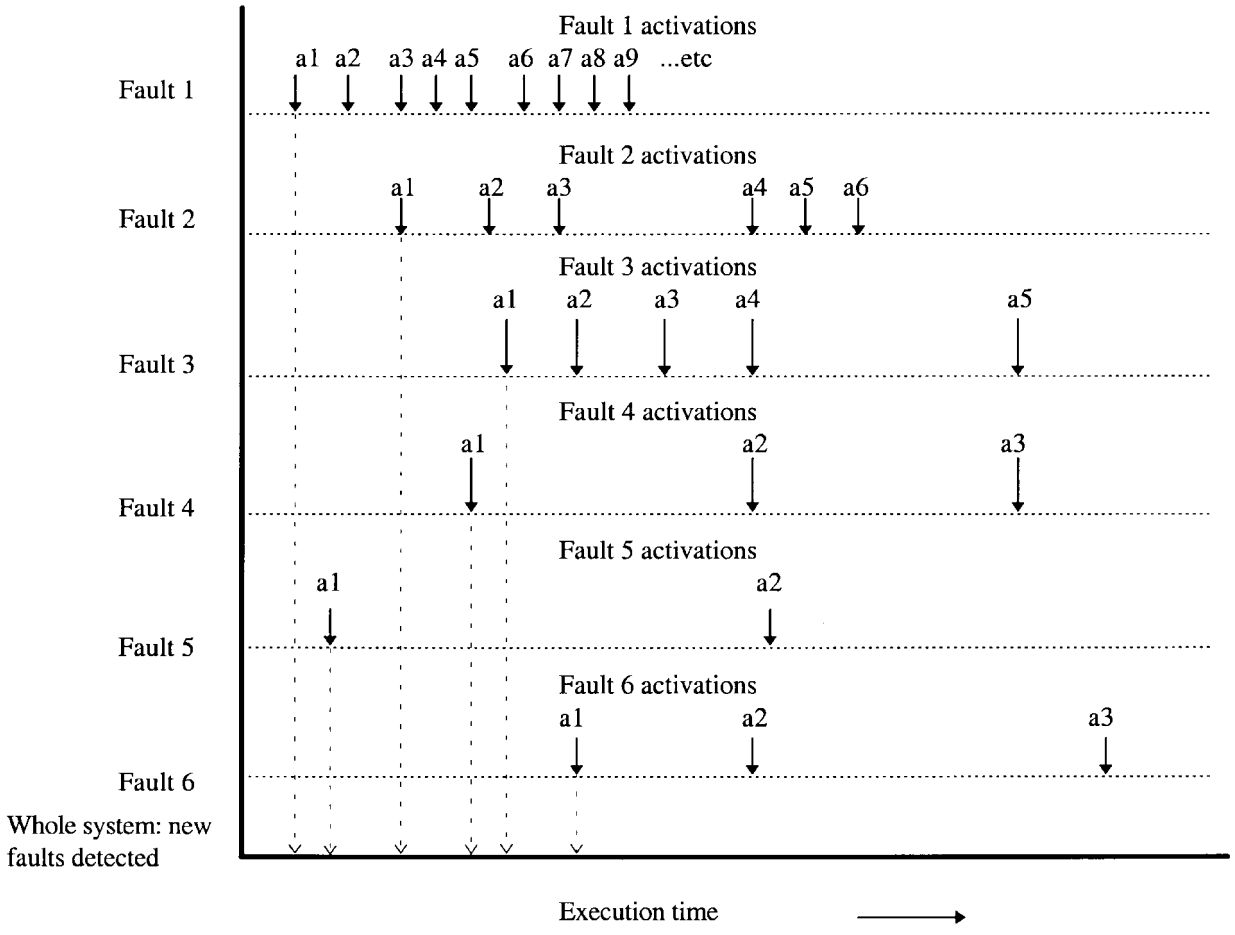


Figure 7. System failure due to activation of latent faults

6.4.12.2 *The Jelinski-Moranda model (JM)*

6.4.12.2.1 *Introduction*

The Jelinski-Moranda [39] is the best known deterministic model. It assumes a fixed (but unknown) number, n, of faults, all with the same activation rate, z. The failure rate of the whole system is proportional to the number of remaining faults, so that, after c faults have been activated and removed, the rate is given by $(n - c)z$. The system failure rate is constant between failures, as illustrated in figure 8.

The JM model therefore has two parameters, n and z, which are estimated from previous failure data using MLE (see **6.4.10**).

6.4.12.2.2 *Data required by the Jelinski-Moranda model*

Like most other fault activation models, JM requires data of the first activation of each fault over execution time.

It was one of the first software reliability growth models to be published. The authors took as their examples two data sets, one from the space shuttle and one from the US Navy. The measure of time that they used was, in fact, calendar time. They state that they would expect better results using an appropriate measure of execution time but that this was not available to them.

6.4.12.2.3 *Estimates provided by the Jelinski-Moranda model*

The JM model can be used to estimate all of the quantities listed in **6.4.1.3**. In particular, since it assumes a fixed number of faults, theoretically it can estimate the expected time at which the last fault will be removed and the software will be 'perfect'.

6.4.12.2.4 *Advantages of the Jelinski-Moranda model*

Because of the simplicity of its assumptions, the JM model is easy to use, and the objective function for the inference procedure to estimate the values of the parameters is simple.

6.4.12.2.5 *Disadvantages of the Jelinski-Moranda model*

In practice it has been found that the estimation of the parameters causes great difficulty, particularly that of n, the number of faults in the program. The main problems are that an optimum value for n does not always exist, so that the search does not terminate. Also the estimated value of n is unstable, so that successive estimates vary widely as the set of failure data grows. In particular the estimate tends to take a value equal to 'the number of faults found so far plus a few'.

The model tends to yield optimistic estimates of the time to next system failure.

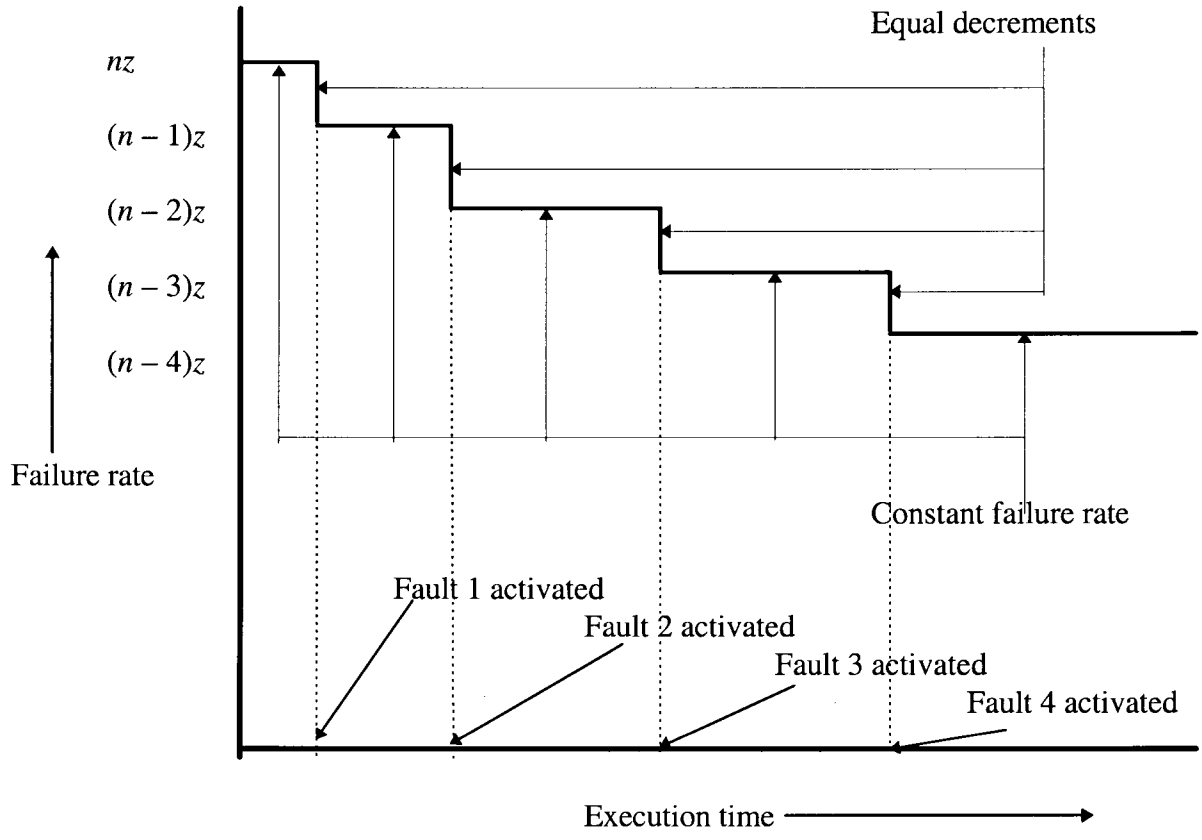


Figure 8. Fault activation and correction in Jelinski-Moranda

The assumption of a uniform manifestation rate for all faults is thought now to be grossly unrealistic. For example Adams [40] reported data collected during the operation of an operating system on a large sample of installations which showed fault activation rates that differed by many orders of magnitude. This incorrect assumption is possibly responsible for the optimistic predictions frequently obtained from the Jelinski-Moranda model (see figure 9). In assuming a uniform manifestation rate for all faults, the model is essentially attempting to fit a straight line to the graph of the system failure rate (i.e. rate of activation of new faults) against the cumulative number of faults activated. If later faults contribute less to the system failure rate than earlier faults, so that the actual shape of the graph is as shown in figure 9, the regression line will decrease in slope as the data set grows, but will always meet the 'rate = 0' axis just ahead of the last data point.

6.4.12.3 *Other deterministic activation rate models*

Several other early models described in the literature are of the deterministic class. Most are of historical interest only but a few examples are outlined below.

a) The Goel-Okumoto imperfect debugging model [34] makes the same assumptions as JM, including uniform fault manifestation rate, but models imperfect fault correction by assuming that a corrective modification corrects a fault with probability p, and fails to correct it with probability $1 - p$. It derives expressions for distribution of time to completely debugged program, distribution of time to a specified number of remaining faults, distribution of number of remaining faults, expected number of faults detected by a given time and system availability.

b) The Shooman model [41] assumes that the failure rate is proportional to the number of faults remaining. The number of faults is normalized with respect to the number of machine language instructions in the program, and so the formulae deal with the number of faults per instruction. Since program size is assumed to be constant, this is only a minor difference from JM. A more significant difference is that the model assumes that the number of faults corrected depends on months of debugging effort, so that reliability is expressed as the probability that a given number of failures will be observed in operating time t, given that e months of effort have been spent on debugging.

c) Schick and Wolverton [42] assume that the failure rate is proportional to the number of faults remaining and to the debugging time since last failure. This implies that the failure rate is linearly increasing between failures, which is rather odd. (The authors also describe a variant of the model in which the failure rate is proportional to a quadratic function of the debugging time. The result is that debugging time between failures follows a distribution referred to as the Rayleigh distribution.)

d) Moranda [43] devised a model to overcome the uniform fault activation rate assumption of JM. The failure rate after a fault correction is assumed to bear a constant ratio (less than 1) to the previous rate, so that the rates decrease geometrically as faults are removed. This is equivalent to assuming a deterministically decreasing set of fault activation rates.

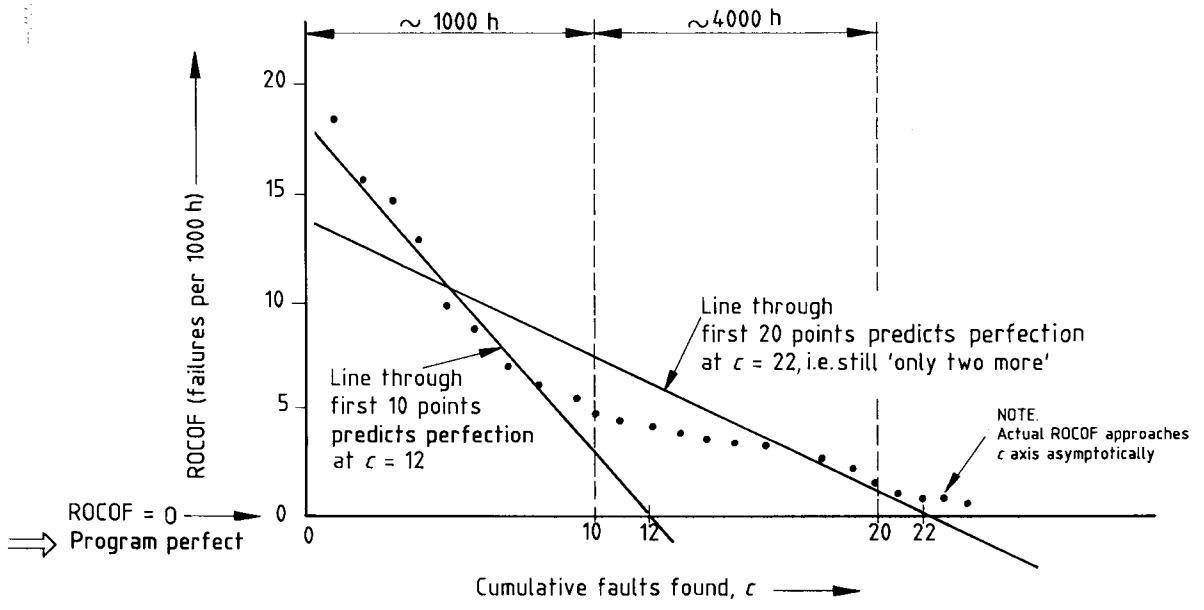


Figure 9. Illustration of why the assumption of uniform fault size leads to optimistic estimates

e) The generalized Poisson model [44][45] is a generalized model of which JM and Schick and Wolverton are special cases. It requires 'failure count' data. Testing is divided into a number of intervals, and it is assumed that faults are corrected only at the ends of intervals. The basic assumption is that the number of faults activated in each interval has a random (Poisson) distribution whose expected value is proportional to the number of faults remaining at the start of the interval and to some power of the time length of the interval. Why detection efficiency should depend on interval length is not clear, but it has been argued that it reflects increased test coverage achieved during a longer interval. If the power parameter is 1, the model reduces to JM. If it is 2, it reduces to the 'quadratic' variant of the Schick and Wolverton model.

f) The binomial model [45] also uses failure count data except that the number of faults activated in each interval follows a binomial distribution with a given probability of activation of each fault. Again the activation rate is uniform over faults. The authors also consider a function other than the usual exponential function as an alternative to represent the probability of activation of an individual fault in each interval (see annex B).

g) Brooks and Motley [46] assume that the number of faults at risk of activation in each test is proportional to the number of faults remaining. It allows imperfect correction by assuming that the number of faults reinserted is proportional to the number corrected. It uses failure count data, and represents individual modules, so that the data consists of the numbers of faults detected in each module in each test. It assumes that faults are distributed homogeneously over the product (a dubious assumption).

It will be seen that most of these models are attempts to refine the JM model by including additional assumptions regarding fault correction, or adapting the approach specifically to failure count data. Most fail to overcome the basic limitation of JM, which is to assume that all faults are activated at the same rate.

6.4.13 *Random activation rate models*

6.4.13.1 *Introduction*

Models in this class assume (like those in the deterministic activation rate class) that the system contains a finite but unknown number n of faults at the start of the period of observation. However instead of assuming that fault activation rates are defined deterministically, these models represent the rates as realizations of a set of random variables. This represents uncertainty about the set of fault activation rates within the system. A consequence is that, during a period of failure-free operation, the failure rate of the whole system can decrease, representing increased confidence that few faults remain or that those that do remain have low activation rates (see figure 10).

Since the number of faults activated in any given interval is binomially distributed, random activation rate models (and deterministic models) which assume a finite fixed but unknown number of faults are sometimes referred to as binomial models.

6.4.13.2 *Littlewood stochastic reliability growth (LSRG)*

6.4.13.2.1 *Introduction*

The LSRG model [47] is a well-known example of this class. It models individual fault activation rates as independent identically distributed random variables. The rates are assumed to follow a probability distribution known as the gamma distribution, which has two parameters, a 'scale' parameter denoted by h and a 'shape' parameter denoted by s.

The model therefore has three parameters, n (the number of faults), h and s. It is capable of representing the fact (noted by Adams [40]) that large software products tend to contain very many faults each of which has a very small individual rate of activation, and that the activation rates differ by many orders of magnitude. The distribution of rates assumed by LSRG is such that if any fault is selected at random it is probable that it will have a very low activation rate and extremely improbable that it will have a high rate.

As the software is executed and debugged, the number of remaining faults decreases and also those that remain tend to be those with smaller activation rates. The formula for the distribution of rates is modified during execution to reflect this (see annex B for the mathematical details).

6.4.13.2.2 *Data required by LSRG*

The parameter values can be estimated from the data by MLE or LSD. Either time to failure or failure count data (see **6.4.3.1.2**) can be used. Only the objective function needs to be rearranged to enable the model to be used with failure count data.

6.4.13.2.3 *Estimates provided by LSRG*

The LSRG model is capable of estimating all of the measures listed in **6.4.1.3**, and a few more besides (e.g. expected individual fault activation rate after a given amount of execution time).

However one exception is that the mean time to failure (MTTF) does not always exist. (The mathematical function which represents MTTF yields a meaningless value for certain values of the parameters). The use of the median value of time to next failure instead of mean time to failure is therefore recommended. This always exists and can be derived easily from the model.

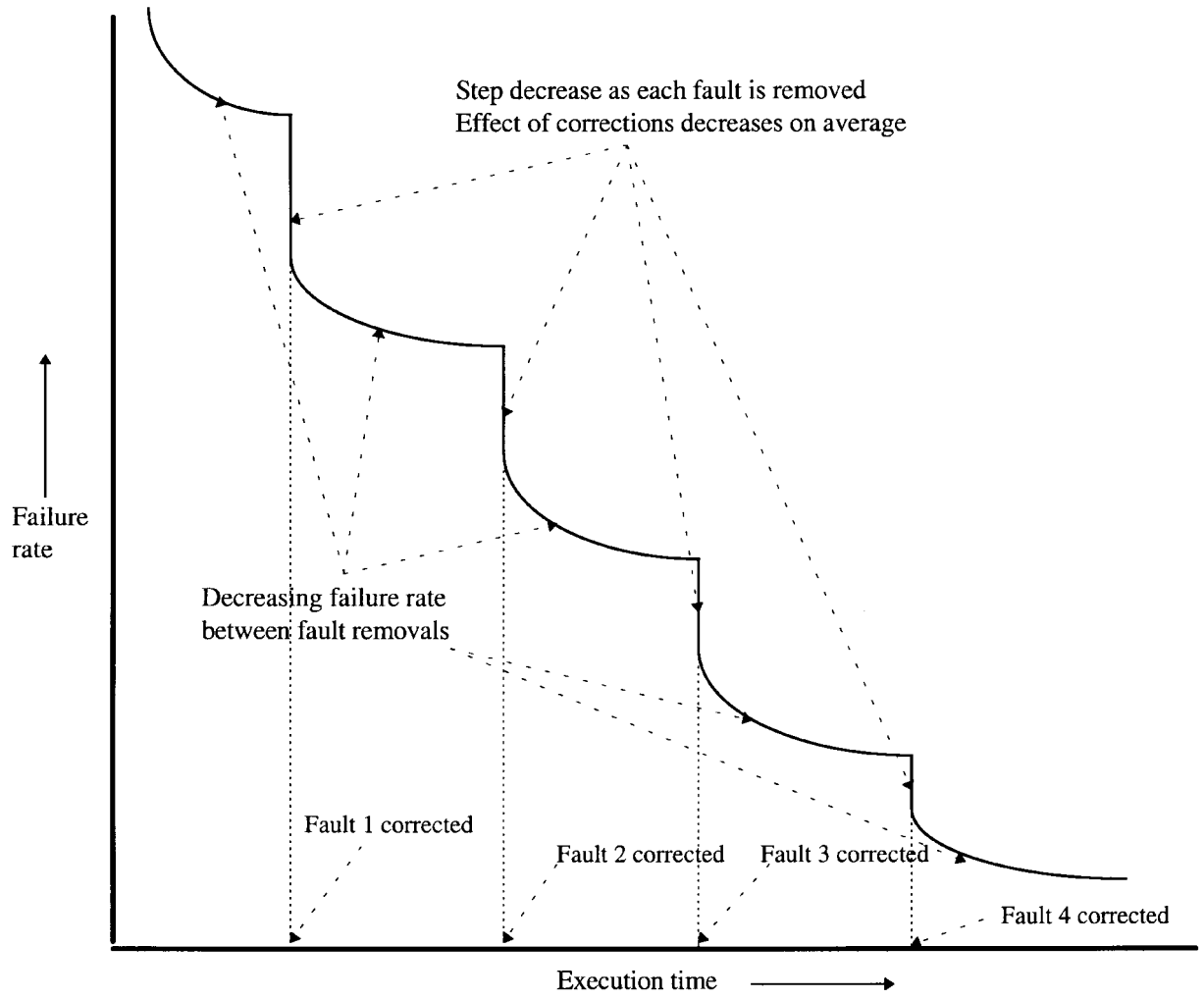


Figure 10. Fault activation and correction in LSRG

6.4.13.2.4 *Advantages of LSRG*

The assumption of the LSRG model regarding the distribution of activation rates is intuitively appealing, and may be expected to correspond well with reality. Although not as simple as JM, it is still highly mathematically tractable. The fact that it has three parameters makes the MLE or LSD optimization slightly more complicated, but mathematical techniques can be used to reduce the number of degrees of freedom for the search to two (see annex B). It is capable of estimating almost all reliability measures that are likely to be of interest.

6.4.13.2.5 *Disadvantages of LSRG*

Despite the improved realism of its assumptions, the LSRG model has still been found to yield optimistic estimates in practice. There is a tendency for the parameter estimates to be either of the 'large n, small s' or else of the 'small n, large s' variety. In the latter case, it is usually found that the estimate of n is 'the number of faults seen so far, plus a few', as tends to happen with JM. Again, assessment of predictive accuracy is required (see **6.4.20**).

6.4.13.3 *Other random activation rate models*

The Weiss model [48] is a hardware reliability growth model whose application to software was suggested by Thayer et al. [49]. It also treats fault manifestation rates as having individual values taken from a probability distribution. It assumes a constant failure rate between failures but allows for imperfect repair by assuming a certain (fixed) probability of a repair action successfully removing a fault.

The Ramamoorthy-Bastani stochastic model [50] makes an equivalent assumption, by treating the size of the decrease in failure rate following a repair (assumed perfect) as being random, with an appropriate distribution.

6.4.13.4 *Random number of faults models*

6.4.13.4.1 *Characteristics of random number of faults models*

Models in this class also generally assume that the individual fault activation rates are randomly distributed, but in a slightly different way. In mathematical terms, they are assumed to be a realization of a non-homogeneous Poisson process (NHPP). Also, the 'number of faults in the system' is treated as a random variable with a certain expected value.

This means that the overall process of system failure is a random process with a rate which varies over time, i.e. it is a NHPP, hence models in this class are sometimes referred to as NHPP models.

It can be shown [51] that any binomial model can be transformed into a NHPP variant by treating the number of faults as a Poisson-distributed random variable rather than as a fixed but unknown quantity. The binomial and NHPP variants will be indistinguishable from the analysis of a single set of failure data.

6.4.13.4.2 *Data required by random number of faults models*

These models all require data of first activations of faults over execution time in either time to failure or failure count format (see **6.4.3.2**).

6.4.13.4.3 *Estimates provided by random number of faults models*

These models can estimate all of the quantities listed in **6.4.1.3**.

6.4.13.4.4 *Advantages of random number of faults models*

Models in this class possess all the advantages of random activation rate models, as well as being slightly more mathematically tractable.

6.4.13.4.5 *Disadvantages of random number of faults models*

As with other classes, caution should be exercised in using these models since they might give optimistic estimates. Some models in this class have an assumption of uniform fault activation rates built into them in a less than obvious way.

6.4.13.5 *Goel-Okumoto model*

The Goel-Okumoto (GO) model [52] is a fairly early and well-known example of its type. It assumes that the cumulative number of faults activated by any given execution time is a random quantity. In mathematical terms it follows a Poisson distribution whose mean value $m(t) = n[1 - \exp(-zt)]$ where n is the expected number of faults that will be activated as execution time t tends to infinity, and z is a constant of proportionality which can be interpreted as the mean activation rate of an individual fault. It will be seen that this is equivalent to assuming a uniform activation rate. Data required, methods of parameter estimation, etc., are similar to other models in this class.

6.4.13.6 *Musa model*

The Musa model [53] is more or less identical to GO in its basic assumptions. However Musa generalizes the basic model in several ways. First, an 'error reduction factor' is introduced to allow for delayed fault correction, and a 'test compression factor' to represent accelerated detection of faults during test as opposed to during service. Execution time is then related to real time by a calendar time model. The development life-cycle is divided into three phases, in which the constraints are the limited availability of fault identification effort, fault correction effort and computer time respectively. Different methods are used in each of the three phases to relate execution time to real time. Sixteen parameters are used to establish this relationship in terms of features of the development organization. It is then possible to make predictions of various reliability measures in real time, so that the model can be used to manage the development process.

As with most such models, the assumptions should be applied with caution, when the model is used outside the organization in which they were devised.

6.4.13.7 *Musa-Okumoto logarithmic Poisson model*

The Musa-Okumoto logarithmic Poisson (MO) model [54] is a recent and fairly representative example of its type. It has two parameters, $r(0)$, the failure intensity at time 0, and z.

The counting process of cumulative faults activated is a NHPP with intensity function given by $r(x) = r(0)\exp[-zm(x)]$ so that the decrement in the system failure rate per fault corrected decreases exponentially with each fault removed.

The model uses the same type of data as others in its class. One advantage is that the parameter z can be estimated by linear regression of the empirical failure rate (estimated from the data) against cumulative number of faults activated.

6.4.13.8 *Other random number of faults models*

Some examples of such models that are described in the literature are the following:

a) a variant of the Duane model [55] assumes that the number of faults activated at any point is a random (Poisson distributed) quantity whose expected value is proportional to some power of the execution time. In addition to MLE or LSD, the graphical approach described in **6.4.5** can also be used to estimate the parameters. The original Duane model is described in **6.4.14**;

b) Rushforth, Staffanson and Crawford [56] devised a random fault number variant of the JM model which uses failure count data. Faults are repaired at the ends of intervals of testing, and imperfect repair is modelled by assuming that only a proportion of faults are corrected, and that new faults may be created;

c) The Schneidewind model [57] is similar, but assumes an exponentially decaying failure rate function. Imperfect repair is modelled by assuming a fault correction rate proportional to, but not equal to, the fault activation rate;

d) Langberg and Singpurwallah [58] adopted an approach that seeks to unify many of the other models described. This is essentially a Bayesian modification to the JM model. It is shown that, if the 'number of faults' parameter in the JM model is treated as a random (Poisson distributed) variable, then the model becomes identical to GO. This is similar to the unification achieved by Miller [59], but the model is also claimed to generalize the LSRG and Littlewood-Verrall models.

6.4.14 *Failure trend models*

6.4.14.1 *Characteristics of failure trend models*

Failure trend models treat either the failure rate or the inter-failure times of the system as these evolve over time as their basic elements, rather than modelling them as consequences of more fundamental assumptions about the activation of latent faults.

They are divided into failure rate trend models (see **6.4.15**) and random inter-failure time models (see **6.4.15.3**).

6.4.14.2 *Data required for failure trend models*

Failure trend models require records of failure over execution time. They are mostly capable of using either time to failure or failure count data except that some specifically require data in time to failure format and the requirement to be able to separate out first activations of each fault is not as stringent.

6.4.14.3 *Estimates provided by failure trend models*

Most of the quantities listed in **6.4.3.1** that are concerned with times to failure or failure rate can be estimated by this class of model. However some of them are not capable of estimating certain measures concerned with faults, e.g. expected number of faults remaining in the system after a given amount of execution time. This is a consequence of the fact that models in this class do not possess a parameter which can be interpreted as the number of faults in the system.

6.4.14.4 *Advantages of failure trend models*

Most of these models are capable of representing imperfect and delayed fault correction without being explicitly enhanced to incorporate additional assumptions. Some of them have been found to have less of the tendency to optimism often found in estimates from fault activation models.

6.4.14.5 *Disadvantages of failure trend models*

The restriction on the type of estimates provided is a slight disadvantage. Although these models are less prone to optimistic estimation, the accuracy of their predictions still needs to be checked (see **6.4.20**).

6.4.15 *Failure rate trend models*

6.4.15.1 *Introduction to failure rate trend models*

These models are fairly simple in their assumptions and are usually based on trends in failure rate observed empirically. The main example is the Duane model.

6.4.15.2 *Duane model*

At its simplest, this model fits a power law to the cumulative failure rate or cumulative number of faults activated. It was originally devised from the observation that graphs of these data tend to follow a straight line when plotted on logarithmic graph paper [60]. The cumulative number $m(t)$ of faults activated by time t is given by $m(t) = at^b$, and the failure rate $r(t) = abt^{b-1}$. A NHPP variant of the model has been applied to software and is briefly described in **6.4.13.8**, where the use of log-linear regression to estimate the parameters a and b is referred to.

The data required is the occurrence of failure over execution time. The model makes no assumptions about the first activations of faults as opposed to other failures. In fact, one of the justifications for the power law is precisely that fault correction is imperfect so that new faults are introduced and repeated failures are observed, without which a law of the form $\log [r(t)] = c - dt$ might be expected.

The main advantages of the model are the simplicity of its assumptions and the ease of inferring the parameter values. The nature of log-linear regression means that later data points are much more heavily weighted in the inference procedure than earlier points. This is both an advantage (early failure data may be less representative) and a disadvantage (the inference procedure is sensitive to outliers).

The inference procedure can detect either reliability growth ($b < 1$) or decay ($b > 1$) and so is useful in preliminary analysis of data (see **6.4.4**).

6.4.15.3 *Random inter-failure time models*

6.4.15.3.1 *Introduction to random inter-failure time models*

Random inter-failure time models treat the times between failure of the system as their basic elements, modelling them as independent random variables. By using a sufficiently flexible time-varying parametric family of distributions to represent these, reliability decay as well as growth may be modelled, automatically taking account of imperfect repair.

Their disadvantages include certain restrictions in the reliability measures that they can estimate, and the fact that they cannot be used easily with failure count data.

6.4.15.3.2 *Littlewood-Verrall model (LV)*

The Littlewood-Verrall model [61] is the best known example of an inter-failure time model. It assumes that following each failure some corrective modification to the system is attempted which alters the reliability, so that the times to failure T_i after the ith fault correction are distributed differently. The T_i are in fact assumed to be independent exponentially distributed random variables whose rates are themselves random variables with a gamma distribution. The model has three parameters; the shape parameter s of the gamma distribution, and b_1 and b_2, which define a family of scale parameters $h(i) = b_1 + b_2 i$ for that distribution. If the modifications on average improve the reliability (although imperfect corrections may occur) then $b_2 > 0$.

Like the LSRG model (see **6.4.13.2**), the LV model represents increasing confidence in the system during periods of failure free operation by predicting a decreasing failure rate between failures, and later corrective modifications cause a lesser decrement in failure rate than earlier ones.

6.4.15.3.3 *Other random inter-failure time models*

The Keiller-Littlewood model [62] is similar to the Littlewood-Verrall except that reliability growth (or decay) is induced on the shape parameter of the gamma distribution of the failure rates, i.e. s is a decreasing function of the failure number i, in order to model reliability growth (or a decreasing function in order to model decay).

The Ramamoorthy-Bastani Bayesian model [50] is a modification of the Littlewood-Verrall model that assumes that the successive failure rates are order statistics from a single distribution instead of being independent. This effectively prevents the model from dealing with imperfect repair which is one of the main advantages of Littlewood-Verrall.

6.4.16 ***Environmental factors models***

6.4.16.1 *Characteristics of environmental factors models*

The definition of reliability includes the phrase 'under given conditions of use'. The reliability of a hardware item depends on such factors as the temperature and voltage at which it is run, and the reliabilities of software products have been observed to depend upon the way in which they are used. For example it is often seen that the failure rate of a software product increases suddenly at the end of its trial when it is released for general use. It is also found that different installations report widely differing levels of reliability for the same software product.

Certain factors such as system loading (number of concurrent processes, number of terminals on-line, etc.), hardware configuration, processor power, and type of user are all thought to alter the perceived reliability of software, sometimes by several orders of magnitude, but it is difficult to separate the effects of different factors and quantify them systematically.

A number of approaches have been proposed to tackle this problem. Essentially they estimate a 'baseline' failure rate which would be observed from the product if all factors had some nominal value and adjust it using a mathematical function which represents the effects of various factors in the environment.

6.4.16.2 *Data required by environmental factors models*

In addition to records of faults activated over execution time, environmental factors models require the data to be broken down by installation, and measurements of the chosen factors for each to be made. (The data may also be broken down over time, with different values of the factors applying to different periods of operation.)

Some of the factors may be simply either present or absent. For example, if an operating system kernel is being studied, then the presence of a database management system (DBMS) running on top of it might be expected to affect its reliability (and this has been observed in practice). The DBMS is either running or not running on each installation at any point in time.

Other factors may be present to varying degrees, e.g. the number of users accessing the system via on-line terminals at any time on each installation.

6.4.16.3 *Estimates provided by environmental factors models*

Most such models estimate the basic failure rate under nominal usage, together with the values of certain weighting parameters which represent the degree to which each environmental factor affects the basic failure rate when it departs from its nominal value. Using the estimated parameter values and the measured values of the factors for a given installation or period of operation, the expected failure rate under the given conditions can be predicted.

Environmental factors models are not primarily concerned with estimating reliability growth, but they are not restricted to dealing with a constant baseline failure rate.

6.4.16.4 *Advantages of environmental factors models*

These classes of model are the only ones which purport to allow estimation of software reliability under varying operating conditions. All others previously described have a stringent requirement that data is collected under operating conditions which will continue to hold over the period (and over the installations) for which predictions will be made.

6.4.16.5 *Disadvantages of environmental factors models*

In practice, the choice of factors that are likely to affect reliability is not easy. There may be very subtle differences in the ways in which software is used on different installations which affect reliability but which are not easy to define. Even if they can be defined, they may prove difficult to measure. There are few published examples of the successful application of this approach.

6.4.17 ***Explanatory variables models***

This approach uses a variation of the Cox proportional hazard model [63]. This was originally devised to model the effect of various factors on the failure rate of some item and has found its main application so far in medicine and biology, where the items are people or animals, the failure rate is the mortality rate, and the factors are treatment received, drugs administered, etc.

Its application to software reliability is only recent. Since what is being modelled is a continuing failure rate, rather than simply the probability of death, a variation of the approach referred to as 'proportional intensity' has been suggested, in which the values of the explanatory variables are used to adjust the failure intensity of the system.

The application of the Cox proportional hazard approach to software reliability was suggested by Nagel and Skrivan [64], and by Dale and Harris [29]. Its use has been investigated by Wightman and Bendell [65]. Studies so far have been hampered by lack of appropriate data. This is still an experimental approach, but should be considered where a single software product shows different failure rates under varying usages.

The mathematical details of the model are described in annex B.

6.4.18 ***System parameters models***

Iyer [66] investigated the effect of the values of certain measurable parameters such as percentage utilization of CPU, rate of transfer of data over input/output channels, etc., on the reliability of large operating system software. Using cluster analysis he was able to define a manageable number (around 10) of operational profiles for the operating system, and correlate these with the observed failure rate. He concluded that the use of the profile to adjust the baseline failure rate did have predictive value.

6.4.19 ***Miscellaneous models***

6.4.19.1 *Types of miscellaneous models*

There are a few black-box stochastic models which do not fit easily into the classification scheme of **6.4.2**. Some of the main types of these are the following:

a) input domain based models;

b) seeding and tagging;

c) availability models.

Many of these models are summarized by Dale and Harris [67] and by Dale [68] (see **6.4.19.2** to **6.4.19.4**).

6.4.19.2 *Input domain models*

Instead of analysing the occurrence of failure over execution time, these models estimate reliability from the proportion of a sample of inputs which cause a software product to fail. There are a number of problems associated with this approach as follows:

a) the validity of the result depends upon the method of choosing the sample. One possibility is to choose the sample at random from the input space. This amounts to a requirement for random testing similar to that required for the correct application of software reliability growth models (see **6.4.1.2** and **6.4.10.1**);

b) since the repair of faults is ignored, reliability growth is not modelled;

c) unless the number of inputs submitted is taken to be a measure of execution time, the measure of 'reliability' obtained does not conform to the usual definition of probability of no failure in a given period of operation. However, on many systems such as a protection device on industrial plant, it is the probability of success on demand which is of interest and for this the number of input cycles may be an adequate measure of execution time;

d) the input sample needs to be large in order to yield a reasonable confidence in the results.

Examples are the Nelson model [69], which requires random input selection, The Duran and Wiorkowski [60] approach, which deals with assessment in the case of no observed failures and the Ramamoorthy-Bastani input domain based model [71], which attempts to assess the probability of correctness of a program with respect to various equivalence classes within the input domain.

6.4.19.3 *Seeding and tagging models*

The seeding and tagging method purports to estimate the number of faults in the product from the number found during testing, by applying statistical methods originally used to estimate animal populations from the sizes of tagged samples captured. No estimate of reliability as defined in this British Standard is provided, since the models take no account of execution time.

The basic statistical method is described in Feller [72] and elsewhere. For example, a sample of fish is captured, tagged and released. A second sample is taken later, after the tagged fish have mingled with the rest, and the proportion of tagged fish in the second catch is taken to be the same as the proportion of the total number in the first catch to the entire population. The estimate is biased, but improved estimators have been suggested by Chapman [73].

Mills [74] suggested inserting faults into a program, and observing the proportion of these seeded faults in the total sample found during debugging. One problem here is that the seeded faults are artificial, and may have a greater probability of being found, leading to an optimistically low estimate. Rudner [75] suggested that two debuggers, A and B, should work independently, and that the faults discovered by debugger A should be regarded as the tagged faults in the sample discovered by debugger B.

6.4.19.4 *Availability models*

Most software reliability prediction models take no account of repair time and its effect on availability on the grounds that when a failure occurs it is often possible to resume operation immediately and carry out diagnosis and fault correction off-line. However availability is of concern to many users of software, particularly those using on-line and real-time systems such as industrial process control, avionics systems, telephone exchanges and on-line financial transaction systems. In these systems, it is the time to restore service after a failure which is of interest, rather than the time to diagnose and correct the fault. Restoration of service may involve reloading the system software and application programs, re-establishing communication with sensors, actuators and terminals, restoring corrupted data files, etc. In many cases this will be automatic and very fast compared to fault diagnosis and repair, but may still be significant, and will vary according to the type of system and mode of failure. There is therefore a need for models which can predict availability by incorporating a realistic restoration time distribution function with an appropriate failure model.

An example of such a model is the Trivedi-Shooman availability model [76] which treats software as having two states (up and down) and represents the transitions between these states. It estimates the probability that the system is up at any given time. Starting with n faults, the system is initially up and goes down when the first fault is activated. It remains down until the fault is corrected, when it reverts to its up state, but with $(n - 1)$ faults, and so on. The model requires data of the estimated failure rate and fault correction rate. The assumptions on which the model is based do not distinguish between time to restore service and time to correct fault.

6.4.20 ***Accuracy of stochastic reliability growth models***

6.4.20.1 *Introduction to assessment of accuracy of SRG models*

The main criterion by which the value of any technique of reliability assessment is judged is the accuracy with which it predicts the future failure behaviour of the system under scrutiny. Stochastic reliability growth (SRG) models are no exception, even though they are the most direct means of assessing software reliability under fault correction and their estimates are required in order to validate the more indirect predictions obtained from process models (see **6.2**) and product property models (see **6.3**). Note that any assessment of reliability is a statement about the future. It states how long the system is likely to operate before future failure, or how frequently failures will occur during future operation.

To check the accuracy of prediction of a SRG model it is necessary to take the estimated distribution of the measure that is being predicted, such as time to next failure, or number of failures that will be observed in a future interval of operating time, observe the realization (the actual value), and see where it lies within the predicted distribution. Where reliability growth is occurring, the problem is to do this systematically and efficiently for a series of many observations as the amount of accumulated data increases.

Several statistical techniques are available for making such assessments objectively [77][78][79]. The basic method is to take the first i observations and predict the i + 1st observations for as many i as the data set allows. Several types of inaccuracy may be found in the predictions. Different tests are used to detect the following different types of inaccuracy:

a) bias: the predicted times to failure or failure rates tend to be consistently optimistic or pessimistic (see **6.4.20.2**);

b) failure to capture trend: the successive predictions do not increase or decrease to the same extent as the observed values (see **6.4.20.3**);

c) other sources of inaccuracy: the predictions may show a greater or lesser variation between successive estimates than the observations (see **6.4.20.4**).

After the several models have been compared, several steps may be taken:

– selection: the model that has given the most accurate predictions so far is adopted to the exclusion of its rivals;

– recalibration: the predictions are adjusted in order to correct the previously observed bias (see **6.4.20.5**);

– combination: the predictions from several models are combined to give a 'consensus' (see **6.4.20.6**).

It is important to note that in practice no single model has been found to give the most accurate predictions for all data sets. The model that predicted the failure rate of system A very accurately yesterday may fail miserably when applied to the data collected from system B tomorrow. Therefore predictive accuracy should always be assessed whenever SRG models are applied.

Most of the methods described in the literature can be used with failure time data and predictions only, but similar methods are being devised for failure count data [80]. Other statistical techniques may be applied to make similar assessments of other types of model, e.g. environmental factors or structural models.

6.4.20.2 *Detection of bias: u-plots*

Probability plots or 'u-plots' are used to detect bias in the predictions of times to next failure, based on the analysis of the preceding times. The following is an informal description of the method (see annex C for a mathematical description).

For each data point the probability distribution of the time to the next failure is estimated. The observed failure time is substituted into this mathematical function giving a number u_i which is the probability that the random variable representing inter-failure time is less than the actual observed value. The point of the 'u-plot' method is that if the estimates are unbiased then the u_i numbers for all of the observations will be uniformly distributed in the range 0 to 1.

This in turn can be tested by a simple graphical method by carrying out the following:

a) sort the u_i values into ascending order;

b) denote the sorted numbers by u_j;

c) draw a graph of the points $(u_j, j/c)$ where c is the total number of u_i values.

If the values of u_i are uniformly distributed, the points will lie on the line of unit slope through the origin. Any bias in the predictions will show as a systematic deviation from that line.

Figure 11 shows u-plots for two models predicting next time to failure on the same data set.

The maximum departures from the line of unit slope (distances d_1 and d_2) measure the bias in the predictions of the two models. Model 1 is pessimistic (it tends to predict times to failure less than those actually observed), while model 2 is optimistic, and more biased than model 1.

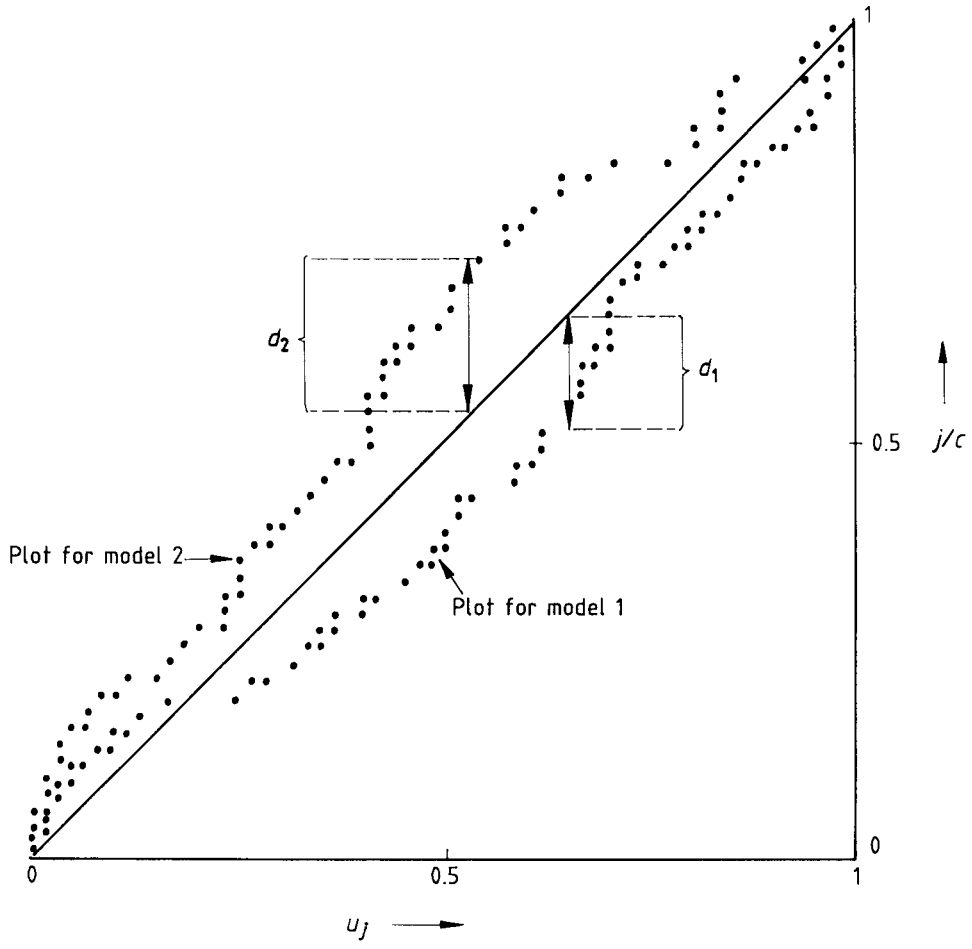


Figure 11. Example of a u-plot: assessment of bias in predictions

6.4.20.3 *Trend capture: y-plots*

A set of u_i values is calculated as defined in **6.4.20.2**. They are further transformed using a logarithmic formula (see annex C) such that if the inter-failure times were predicted accurately the resulting set of numbers can be treated as if they were the inter-event times of a random process with a constant rate. Trend in the estimates will show itself as a non-constant rate for this process. A graphical method similar to the u-plot can again be used to test for this. Deviation from the line of the unit slope in this case reveals trend in the predictive accuracy.

Alternative methods of testing for trend in the transformed u_i values are the Laplace test for trend, or drawing the scatter plot of u_i against i and observing any grouping of points.

6.4.20.4 *Other sources of inaccuracy: prequential likelihood*

This technique is fully described by Dawid [81].

It derives a number referred to as the 'prequential likelihood' for the set of predictions of times to failure. This is defined as the product of a number of terms, one derived from each predicted distribution and observed value.

Suppose that two models A and B are being used, and these have prequential likelihood's P_A and P_B respectively for the given data set. The prequential likelihood ratio (PLR) between the estimates produced by the two models is given by P_A/P_B. It can be shown that, if this ratio tends to infinity as the set of observed data grows, then model A is discredited as a predictor relative to model B. The PLR can therefore be used to judge between each pair of models being applied, and it can be interpreted as the odds that a rational gambler would offer on one model being true against the other, given the observed data set. This is a good test for undue noise in the estimates, as well as other sources of inaccuracy. Other tests for noise include the Braun statistic, median variability and rate variability tests [77][79].

6.4.20.5 *Recalibration*

Predictions can be recalibrated to correct for bias revealed by the u-plot. The technique is referred to as 'adaptive modelling' [78][82] and can be used to adapt the predictions from a model which has a good y-plot but poor u-plot on the data set in question, i.e. is exhibiting a stable bias in its estimates.

The basis of the method is to construct a u-plot at each stage as the data set grows, and to use the actual shape of the u-plot to transform the estimates. The whole procedure is repeated for each observation, with a new u-plot being calculated at each stage.

It should be noted that the adapted predictions are true predictions in that they depend solely on previous data. The adaptive procedure has been found effectively to remove bias from predictions of time to next failure, and when applied to the estimates from several models which disagree, it tends to bring their estimates into closer agreement.

6.4.20.6 *Combination of model predictions*

Instead of using the measures of predictive accuracy to adapt the predictions of a single model (see **6.4.20.5**), it is possible to combine the predictions from several models using appropriate weighting factors.

One approach suggested [29][67] is a technique used in actuarial work and known as 'credibility theory'. Suppose that estimates of some quantity such as failure rate have been obtained using two different methods. The two estimates r_1 and r_2 are combined using the formula $wr_1 + (1 - w)r_2$ where the weighting factor w is referred to as the credibility factor, and represents the confidence of the user in estimate r_1 as compared to r_2. It should be noted that w, and the resulting preference of one estimate over the other, will usually change over time as more information is acquired.

Estimating w presents a problem. If r_1 and r_2 were obtained from different reliability growth models, then a possible approach would be to derive w from the prequential likelihood ratio between the two models [83], which can be interpreted as the odds of one model being true, compared to the other (see **6.4.20.5**).

It has also been suggested [29] that r_1 may be an estimate based on a process model or product properties model, and r_2 may be an estimate based on a stochastic reliability model. As more failure data is acquired, w would decrease to represent the shift in confidence from r_1 to r_2.

6.4.21 ***Structural models***

6.4.21.1 *Introduction to structural models*

6.4.21.1.1 *Characteristics of structural models*

Structural models combine the levels of reliability of individual components to produce an estimate of the reliability of the whole system. The expected levels of reliability of complete hardware systems have been estimated in this way for some time. Similar approaches have been applied recently to estimate the reliability of software from information about the levels of reliability of its individual modules, and to estimate the reliability of a complete system from information about the levels of reliability of its individual hardware or software components.

6.4.21.1.2 *Data required by structural models*

The reliability levels of the individual components should be estimated. One possibility is the use of black-box models at the software module level. In order to combine the module reliability levels, information is required about the proportion of total system execution time spent executing each component, referred to as an execution profile. In practice, some type of code instrumentation is needed in order to measure this.

6.4.21.1.3 *Estimates provided by structural models*

Structural models estimate the failure rate of a complete system obtained by integrating a set of components. They do not generally estimate reliability growth.

6.4.21.1.4 *Advantages of structural models*

Structural models complement black-box stochastic models by allowing total system failure rate to be predicted given that the individual component failure rates are known. This approach could be used to predict system behaviour prior to integration where the components have already undergone individual trials and their levels of reliability have been estimated. It might also be useful where software components whose reliability levels are known from previous service history are to be reused.

6.4.21.1.5 *Disadvantages of structural models*

The estimates are crucially dependent on the execution profile assumed. In practice this is difficult to measure and has been found to depend in turn on the operational profile of the system. In fact, one reason for the sensitivity of software reliability to usage is that usages affect the extent to which different modules are executed.

6.4.21.2 *Parameter estimation for structural models*

There are four aspects to the estimation of parameters for most structural models, as follows.

a) Failure rate of each individual module.

This may be measured by applying a black-box stochastic reliability growth model to each individual module. This in turn requires the first activation of each fault in the module to be recorded together with the amount of execution time it has undergone for a representative period of operation. The resulting data is then analysed statistically.

b) Average execution time in each individual module.

This requires the execution time spent within each module to be recorded for a number of calls to the module over a representative period of operation, and the average to be calculated from these statistics.

c) Frequency of transition across the interface between every pair of modules.

d) Probability of failure on transition across the interface between every pair of modules.

This requires the identification of system failures due to interface faults.

Before recording these, it is necessary to define the level of granularity, i.e. the sizes of the modules into which the system is decomposed, and what actually constitutes a 'module' for the purpose of the investigation.

Items b) and c) above require code instrumentation for their measurement, i.e. the software should be executed on a test-bed or with probes compiled or linked into it in order to record execution times by automatically reading the hardware clock. This will affect the performance of the system, but it should be possible to estimate at least the average of execution time in a module relative to the total software execution time. The average execution time can then be scaled to allow for any improved overall system performance after removal of the probes.

6.4.21.3 *Modular software models*

Modular software models represent the transfer of execution between software modules. This is generally treated as a random process. The main example is the Littlewood structural model, described in **6.4.21.4**.

6.4.21.4 *Littlewood structural model*

6.4.21.4.1 *Characteristics of the Littlewood structural model*

The Littlewood structural model [84] makes the following assumptions:

a) the system consists of a finite set of discrete modules;

b) failure within each module occurs as a random (Poisson) process;

c) each module has its own failure rate;

d) at any time, system execution occupies one module;

e) system execution is transferred between modules at random (in mathematical terms, according to a semi-Markov scheme);

f) each interface between a pair of modules has a given probability of failure.

Two conditions need then to be assumed:

1) system behaviour is observed over a period of time which is long compared to the average sojourn time within each module, i.e. the time which system execution spends within a module before transfer to another;

2) the module and interface failure probabilities are low (as they should be during later integration testing and trial).

Given these two conditions, the failure process of the total system can be shown to be a random (Poisson) process whose rate is given by a fairly simple expression involving the module failure rates and the average execution time spent in each, and the probability of failure on transfer and frequency of transfer across each interface.

6.4.21.4.2 *Data required by the Littlewood structural model*

This model requires all of the four types of data listed in **6.4.21.2**a) to d).

6.4.21.4.3 *Estimates provided by the Littlewood structural model*

The model estimates the failure rate of the whole system, assumed to be constant on average over a long period.

6.4.21.4.4 *Advantages of the Littlewood structural model*

Like most structural models, it allows software component reliability levels to be combined to give a measure of the reliability of the whole system. The expression derived for the system failure rate is simple in form.

6.4.21.4.5 *Disadvantages of the Littlewood structural model*

a) The model is rather abstract and theoretical. In reality, software systems do not usually transfer control between modules at random, but proceed by invocation of one module by another, followed later by return of control from the module invoked to the calling module.

b) Despite the simplicity of the mathematical expression for the failure rate of the whole system it involves a large number of parameters ($2n^2$, where n is the number of modules). Although this might be reduced by the fact that certain pairs of modules never call one another, the practical application of the model is difficult.

c) The probability of failure of the interface on transfer does not need to be treated as a separate parameter, since transfer of control can be regarded as part of the function of the calling module.

The practical problems posed by instrumenting code in order to measure the execution profile, and the effect of operational profile on execution profile have already been mentioned above.

6.4.21.5 *Hierarchical structural models*

A number of published models treat software as being constructed from a strict hierarchy of modules such that any module may be called only by one module in the layer above (which 'owns it') and may call in turn several modules in the layer below. Following successful completion of its function each module returns control to its 'owner'.

One example [85] is similar to the Littlewood model but mathematically simpler due to the hierarchical assumption. The execution profile is characterized by a set of branching probabilities representing control flow down the hierarchical tree, and the overall behaviour of the software can then be modelled as a random process of transfer among a number of states (a 'Markov process') i.e. several 'transient' states corresponding to the execution within any one of the modules, plus two 'absorbing' states representing 'successful completion', and 'failure'.

The reliability of the software is the probability of reaching the 'successful completion' state. This model is more applicable to software which works in discrete cycles, each with a defined stopping point, rather than continuously operating software such as an operating system.

The assumption of a strict hierarchy is restrictive. Real software is often designed in such a way that low-level 'general purpose' subroutines may be invoked by any other module at any level, so that there is no hierarchy.

6.4.21.6 *Simplified structural models*

Despite the problems due to the microscopic detail of the published structural models, it is of value to take account of the structure of software at the macroscopic level. Instead of starting with probabilities of transfer and distributions of sojourn times, it is possible to deal directly with proportion of total execution time spent in each module, measured over long periods of execution.

The simplified model requires the following intuitively appealing assumptions.

a) Each module generates failures at its own rate.

b) Total system execution time is the sum of the individual module execution times, each of which is small compared to the total.

c) The failure process of the whole system is the superposition of the individual module failure processes.

d) If these are random then so is the overall failure process, and its rate is the sum of the rates of the component processes, weighted by the proportion of execution time spent in each module. (This corresponds to one term in the formula of the Littlewood model).

The execution profile is then characterized solely by the proportion of time spent in each part of the software. Such a model requires only very weak assumptions, and no data of interface failure rates are required. Its simplicity renders it more widely applicable than several published models, and it can be extended easily to parallel systems, shared code, etc. It is also not necessary to assume that any part of the system has a constant failure rate.

6.4.21.7 *Hardware/software structural models*

6.4.21.7.1 *Introduction to hardware/software structural models*

The nature of a 'module' in the structural models is loosely defined, and there is no reason why hardware components should not be dealt with using a similar conceptual framework. To do this it is usually necessary to loosen certain assumptions, e.g. that execution only takes place within one module at any point in time, in order to allow for parallel operation of components.

6.4.21.7.2 *X-ware models*

A number of recent models, e.g. [86][87][88], deal with hardware/software systems, reliability growth within structural software and the effects of fault tolerant software design.

These approaches have removed many of the restrictions of the earlier models. In particular they incorporate software reliability measurement within overall system reliability assessment.

6.5 Assessment of high reliability for software

6.5.1 *Introduction to high reliability assessment*

The problem of assessing very high reliability in software is that the observation of any failure during test indicates that the product is inadequate. Methods are required that can yield an assessment on the basis of no observed failures during trial. This observation on its own provides only limited confidence in the reliability of the product, however. Other sources of evidence should be used to assess high reliability software.

Several techniques are described in this British Standard for the assessment of reliability of software with 'normal' requirements. These are classified as follows:

a) Software development process models (see **6.2**).
   1) Inspection statistics (see **6.2.2**).
   2) Qualitative assessment of good practice (see **6.2.3**).
   3) Formal methods (see **6.2.4**)

b) Software property models (see **6.3**).
   1) Software science (see **6.3.2**).
   2) Complexity measures (see **6.3.3**).
   3) Quality factors (see **6.3.4**).
   4) Fault tolerance (see **6.3.5**).

c) Stochastic reliability models (see **6.4**).
   1) General statistical techniques (see **6.4.3**).
   2) Black-box parametric models (see **6.4.4**).
   3) Structural models (see **6.4.5**).

Recent work [2][89] has cast serious doubt on the ability of all of these approaches to provide adequate confidence that a very high level of reliability has been achieved. Their shortcomings are described in **6.5.2** to **6.5.4**.

6.5.2 *Assessment of high reliability from process data*

The argument from 'good practice' is based upon experience with the application of particular development methods or design techniques to previous similar products and knowledge that in those cases a very reliable product resulted. Unfortunately this evidence is rather weak for the following reasons.

a) It is not easy to judge the degree of 'similarity' of the earlier products to the present one. By definition, the comparison will be made with products which are several years old. Both the type of requirement (even within the same application domain) and the available development methods are likely to have changed.

b) Even given a high degree of 'similarity', the fact that a given process was successful in the past gives only limited confidence that it will be successful in the current product. There are many other factors which may confound such a judgement, e.g. skill of personnel, difficulty of current task, 'accidental' factors such as sickness or staff turnover which adversely affect quality, etc. Given n successful trials of a method and in the absence of other evidence, the best prediction is that the method has a probability $p = (n + 1)/(n + 2)$ of succeeding again [89]. Since n is likely to be small, so is the level of confidence.

c) Specifically regarding the use of 'proof of correctness', this may not yield the desired level of reliability for the following reasons.

   – If the requirements specification does not capture the 'real' requirements, the system may not perform adequately in service. Studies have found that the proportion of software faults in safety-critical systems that arise from problems in defining or understanding requirements may exceed 70 % [90]. A proof of correctness against such incorrect requirements would not be helpful.

   – The application of proof of correctness requires a high level of expertise on the part of the development team and a large amount of effort. A written proof is around 10 times the length of the source code being proven. The likelihood of a mistake in the proof is significant.

   – Unless the compiled object code is the subject of the proof, there is a probability that a fault in the compiler may result in a fault being inserted in the compiled code although the source code has been proven 'correct'.

   – The system as a whole might fail due to the software not interacting correctly with the hardware because the hardware platform on which the software is executed may itself contain design faults.

d) The use of inspection statistics to derive a measure of residual 'fault density' in the delivered software also depends on comparison with 'previous similar products' and so is subject to similar problems to those described in a) and b) above. Density of delivered faults is only an indicator of reliability. Although there is no doubt that inspections have a beneficial effect on software reliability, safety-critical software which has been subjected to careful inspection has still been found to contain faults which were activated in service.

6.5.3 *Assessment of high reliability from software properties*

It has been proposed that evidence of 'good design', and specifically the use of fault tolerance, may be cited in support of a high reliability assessment. This argument fails for the following reasons.

a) Little evidence is available on which to base a correlation of 'good structure' or 'low complexity' with high reliability. This is an instance of the more general problem of lack of sound experimental evidence for the efficacy of accepted software engineering practices, although it may seem intuitively obvious that these practices are a 'good thing' and are likely to improve software quality. As a result evidence of 'good structure' or 'low complexity' can give little confidence in themselves that high reliability will be achieved.

b) Specifically where the use of fault tolerant software design is concerned, there is good evidence that independently written versions of a software module do not fail independently [91][92][93]. This seems to be due to the fact that certain parts of any requirements specification are 'difficult' and all the developers are likely to make mistakes in these areas (and hence introduce similar faults). As a result (although it is well established that use of fault-tolerance does improve reliability) such designs should not be relied upon to provide very high levels of reliability, since the probability of a common-cause failure is sufficiently large to swamp the extremely low probability of failure required.

6.5.4 *Assessment of high reliability by stochastic models*

The basic problem with assessing very high levels of reliability based on stochastic reliability models is that such models require a data set containing a fairly large number of fault activations in order to yield estimates within reasonable confidence limits. Unfortunately, the detection of a number of faults during trial or operation is strong evidence that the system is insufficiently reliable, whereas conversely the observation of a long period of failure-free operation constitutes only weak evidence of high reliability.

It has been shown [89] on the basis of a Bayesian analysis that, given no other information about the product, the time of failure-free operation which has been observed is the median of the residual time to first failure, i.e. if a product has operated for 1000 hours without failure, there is only a 50/50 chance that it will continue to operate successfully for a further 1000 hours. In order to arrive at high reliability after a period of failure-free testing, it is necessary to start with a similarly high prior belief, e.g. based on expectation from previous experience that the process or design used will yield high reliability.

A similar analysis [94] has shown that, under random testing (representative of actual use), to achieve a 99 % confidence that the failure probability is less than 10^{-9} per input case requires 4×10^{10} test cases. In considering the effect of prior belief in reliability Miller cites an example in which the prior belief is that there is a probability of 0.01 that there is a fault in the product with a manifestation rate of 10^{-6} per input case. This means that the prior mean probability of failure is 10^{-8} per test case. He points out that to give a posterior probability of 0.99995 that no failure will occur in 10^{-8} input cases still requires 10^{-7} failure-free test cases.

In other words, using the Bayesian approach, if the a priori belief is that the product is 'good enough', no data are needed, but if it is slightly less than good enough, a very large test sample is required.

6.5.5 *Conclusions on assessment of very high reliability*

There are two distinct but related problems with very high reliability: how to achieve it and how to know that it has been achieved. For software neither of these problems is readily resolved at present.

It is possible to produce software based systems which are of adequate reliability for most applications. However, the state of the art does not permit the production of software which can justifiably be claimed to exhibit a very high level of software reliability. This does not mean that software based systems cannot be produced for which very high reliability can be claimed for the system.

The production and assessment of high integrity systems containing software is an issue on which much guidance is becoming available. References [5] and [6] are examples. The reader should consult these sources for further guidance.

From the point of view of reliability the best recommendation that can be made is that the aim should be to design systems so that it is not necessary to claim high reliability for the software. In particular, a software component should not be allowed to provide a single point of failure in a system for which high reliability is required.

7 Application procedures

7.1 Introduction

7.1.1 *Overview*

Application procedures are referred to briefly in **5.9**. This clause details actions which should be carried out in order to apply the methods described in **6.1** to **6.5**, and covers the following aspects of the procedures.

a) Types of data that should be collected for use with the various methods of assessment.

b) Procedures for the collection of raw data so that meaningful measurements can be made.

c) Database structures for the storage of raw software failure data, and methods of extracting direct measurements.

d) Various procedures for the maintenance of software and recommendations on their relative effectiveness.

The principles of measurement as they apply to software reliability are summarized in **4.4**, and the purposes of measuring software reliability in **5.2**. Direct measurements required for each method of software reliability assessment are summarized within the description of the method in **6.1** to **6.5**. These subclauses detail the minimum raw data required, particularly the data needed to apply stochastic reliability models (see **6.4**).

7.1.2 *Fundamental principle of data collection*

Data should be collected with a clear purpose in mind, and with a clear knowledge of the ways in which they will be analysed in order to yield the desired information.

The methods of assessment to be applied should be chosen in advance of setting up any data collection programme. The measures that are to be evaluated should be defined precisely. These will usually be indirect measures. The direct measures from which they are derived should be determined from their definitions. These in turn will determine what raw data is required.

7.1.3 *Means of data collection*

Data may be collected in one of two ways.

a) On paper forms which are completed by development personnel or by users of the software.

b) Using automatic facilities such as those provided by CASE tools, compilers, editors, etc., during development, and by software instrumentation or by recording facilities in the operating system during trial and operation.

The use of paper forms requires effort on the part of the data providers, and renders the data collection procedure susceptible to human error, to incomplete reporting due to pressure of work and to falsification of data for various reasons. Good form design may alleviate these problems. The resulting measurements should be fed back to the data providers in order to motivate them.

Some types of data, e.g. inspection statistics, can only be collected manually, and automatic data collection facilities may not be available in other cases.

Data should be collected automatically whenever possible. Some types of data, e.g. module execution times for the extraction of an execution profile (see **6.4.21.1.2**), cannot be collected manually.

A central data collection function (CDCF), should be set up to coordinate the collection of data. Its responsibilities are as follows.

a) Ensuring that all data required is provided. This may involve chasing human data providers and ensuring that the contents of automatic data repositories are gathered.

b) Checking that paper forms are correctly completed and referring back to the originators any that are in error. It has been found that, without such a check, a substantial proportion (as high as 50 %) of forms may be incorrect.

c) Enforcing formats of reporting. The data to be collected should be defined consistently across the company so that different projects may be compared. Specified formats should be used for dates, times, identifiers, etc.

d) Entering data into the company database and maintaining its integrity.

e) Giving feedback to data providers.

The size and structure of the CDCF will obviously depend upon the size and structure of the company or other organization which is collecting the data and upon the type of data being collected.

7.1.4 *Means of data storage*

Software reliability data is an important part of corporate memory (see **5.2**). To facilitate data extraction and analysis, an electronic database should be used where possible. Its structure will depend on the methods of assessment chosen and the data needed for these, and no all-purpose data model can be defined. However, in the specific case of software failure data for stochastic reliability modelling, a basic database structure can be recommended (see **7.4.7**).

7.1.5 *Configuration management*

It is necessary to be able to identify clearly each software entity whose reliability is to be measured. In the course of development, each sub-system or module will be modified repeatedly, and several versions of each component will exist. It is important to know which version of each component is included in a given version of the total system. Configuration management is the set of procedures and practices concerned with keeping track of the modification level of the system and its components. Proprietary CASE tools may be used to assist in this task. (For a system including both software and hardware, the hardware modification level should be identified also.)

The formalized methods for retaining configuration control of computer-based systems are described in BS 6488, and these procedures should be followed.

It is essential to know which version of a system has been released for use, and to ensure that it incorporates the appropriate version of each component and the corresponding version of documentation. An important consideration is that the reliability of a system will usually change when it is modified, so that any measurement of the reliability of a system applies to a specific version. However if every small change resulted in a new version which was treated as if it were a new product, this would preclude the use of reliability growth models. It is therefore necessary to define a baseline version whose identity only changes following a substantial modification. Effectively the version identifier has two parts, the baseline identifier and a further identifier for the detailed modification state. It is important that the identification of baselines is kept simple to permit easy referencing, e.g. in failure reports.

Engineering judgement should be used to determine what constitutes a baseline change. The modification of a large part of the software or the addition of new modules to enhance its function would almost certainly necessitate a new baseline. The correction of a few faults while retaining the original functional specification and software architecture would be unlikely to justify changing the baseline.

Configuration management CASE tools control the source code for each module, record all faults reported and all changes made, assign and update version identifiers, and construct build lists for system compilation and integration. They are therefore useful sources of automatically recorded data.

7.1.6 *Software item data*

All software items whose levels of reliability are to be assessed should be identified. Software item data is fundamental to reliability assessment based on software properties or on stochastic reliability models, and is also important for assessment based on process models. A software item may be any of the following.

a) System: a complete software entity such as an application program, utility program, tool, operating system, embedded control program, etc. It is free-standing in that it is not essentially part of a larger system (although it will have interfaces to other hardware and software items). It may be regarded by the developer as a separate commercial product.

b) Sub-system: a self-contained part of a larger system, with a defined function and a defined interface to other sub-systems. For some purposes a sub-system may be regarded as a system in its own right, and may consist of smaller sub-systems. A system may therefore have a hierarchical structure, consisting of many levels of sub-system.

c) Module: the smallest self-contained unit of software. A module is the lowest level of sub-system. It may possess its own internal structure but is regarded as atomic for purposes of reliability assessment. The level of structure below which the system is not further decomposed depends on the level of detail at which it has been decided to perform the assessment.

For some types of software, e.g. object-oriented, client-server based, window-based, etc. the parts may be described differently, e.g. 'class of object' may be used in place of sub-system or module.

Certain documents, e.g. functional specifications, user manuals, etc., are part of the software product and should be recorded as such, with appropriate identifiers, version numbers, titles, etc. Documents may be regarded as 'sub-systems' for recording purposes.

Any software is executed as part of a larger hardware/software system. Where reliability of a complete system is assessed similar data should be recorded for hardware items.

The following data should be collected for all software items.

1) Identifier: a unique means of referring to the item.

2) Baseline version identifier (see **7.1.5**).

3) Minor modification level identifier (see **7.1.5**).

4) Name and function: a readable description of the item.

Other data recorded for software items will depend on the type of system, on the chosen methods of assessment and on the indicators selected for comparison in order to establish which are significant for reliability. Several types of data should be considered for analysis such as the following:

– severity of consequences if item fails during operation, e.g. minor inconvenience to user, possible loss of human life, major economic losses;

– size, e.g. in the case of program code, lines of code;

– structure, e.g. the degree of conformity to accepted or pre-defined notions of good structure;

– complexity, (see **6.3.3**).

7.2 Procedures for use with process models

7.2.1 *Introduction to process model procedures*

Software development process models (see **6.2**) yield estimates of the reliability of the delivered software based on the evaluation of certain indicators during development. These may be measurements of intermediate products such as statistics collected during inspections (see **6.2.2**), or of an assessment of good practice. (see **6.2.3**).

The overall procedure should be as follows.

a) Define in advance the indicator measures which might be expected to correlate with reliability.

b) Design data collection forms for use by development personnel.

c) Train personnel in the use of the forms and ensure the CDCF is ready to receive and check them.

d) Analyse the data collected and feed back the measurements to the development personnel.

e) Measure the actual level of reliability of the delivered software in trial or operation in order to establish the degree of correlation with the chosen indicators.

7.2.2 *Procedures for use with inspection statistics*

The procedures for carrying out inspections, recording instances of non-conformance, and analysing the statistics are described in **6.2.2**. Other than the general recommendations for application procedures in **7.1**, no further recommendations are needed.

7.2.3 *Procedures for assessment of good practice*

In order to evaluate the effectiveness of a particular process, it is necessary to know what methods were used and how rigorously they were applied. Some items of process data which could be recorded are as follows:

a) the development method, e.g. name of structural design technique;

b) the development control, e.g. quality assurance procedures used;

c) the development mode, e.g. host/target working;

d) the testing approach, e.g. strategy, techniques, method and type.

It is beyond the scope of this British Standard to make detailed recommendations regarding all of the development practices that may be recorded, since these are too numerous. The forms used for data collection and the structure of the database used for storage are dependent on the practices selected for recording, and similarly no detailed recommendations can be made.

7.3 Procedures for use with product property models

Product property models (see **6.3**) base their estimates of reliability on the evaluation of indicators. The general recommendations in **7.2.1** therefore apply.

Various proprietary CASE tools are available to assist in the automatic evaluation of quantities such as program size and complexity (see **6.3.3**). Static analysers can display the structure of the software as a flowgraph and assist the user in detecting undesirable features of this structure. Test-beds can automatically record the coverage achieved by a given set of test cases (see **6.3.3.3**).

Software properties should be recorded as part of software item data (see **7.1.6**).

7.4 Procedures for use with stochastic reliability models

7.4.1 *Introduction to stochastic reliability model procedures*

To estimate software reliability using stochastic reliability models the following types of data should be collected.

a) Products: the identity and baseline version of each software item to be assessed (see **7.1.6**).

b) Installations: A record of each physical set of hardware equipment on which a copy of the software to be assessed is being executed.

c) Failures: A record of every occasion on which the system departed from its required behaviour (see **4.3**).

d) Faults: A record of every software fault which has been detected (see **4.3**).

e) Changes: A record of every modification made to each software item either to remove a fault or for other purposes (see **4.3**).

f) System use: A record of the amount of use of each software item on each installation.

These data should be collected during a trial under a realistic operational profile (see **4.5.4**a)) and during operation. The overall procedure should be as follows.

1) Identify the software items to be assessed. Ensure these are under configuration control and define their product and baseline version identifiers (see **7.1.6**).

2) Identify all installations from which data is to be collected and define means of referring to them unambiguously.

3) Define the reliability measures to be assessed (see **6.4.1.3**).

4) Define forms and procedures for recording failures, train data providers in their use, and ensure CDCF is ready to process them.

5) Define meaningful measures (see **7.4.6**) of software use and procedures for recording this on each installation.

6) During trial or operation, CDCF receives failure reports from the installations. Each report is recorded on the database and passed to the support team for diagnosis.

7) The support team diagnoses the fault responsible for the failure, devises a change to correct the fault, and responds to CDCF with this information.

8) CDCF records the fault and the change in the database, inserts a cross-reference from the failure record to the fault record, and responds to the originator of the failure report at the installation.

9) CDCF collects records of software use from all installations and enters these into the database.

10) After a certain period of trial or operation, time to failure or failure count data (see **6.4.3.1.2**) are extracted from the database. These are analysed using stochastic reliability models (see **6.4**) to provide estimates of the chosen measures of reliability.

The above is a broad outline of the procedure. It should be adapted to particular circumstances of trial or operation and to different types of software product as necessary. Some ways in which this may be done are described in **7.4.2** to **7.4.6** where the data to be collected is described in more detail.

7.4.2 *Recording installations*

An installation is a self-contained set of hardware equipment on which one or more copies of the software being assessed are executed. The following data should be recorded for each installation.

a) Identifier: a unique code used to refer to the installation.

b) Geographical location: the address of the site holding the installation.

c) Contact name: an individual on the site with whom CDCF may communicate.

d) Dates on which the hardware equipment was commissioned, modified, and decommissioned.

e) Dates on which the software being assessed was delivered for trial or operation, modified, and withdrawn from service.

f) Configuration: the current hardware and software environment should be identified including the make, model and modification level of all processors and peripherals, and of the operating system and other software used in conjunction with that being assessed. Any changes to the configuration should be recorded along with the dates and times at which they are effective. This information may assist diagnosis of the causes of failure.

This applies where software is in use on a number of fixed installations. Other scenarios may require the adaptation of the data to be collected.

1) Bespoke software: this is developed under contract for a specific customer and may be used on a single installation.

2) Distributed systems: these require special care in recording the amount of use of the software, since many users may have simultaneous access to it, and copies of the software itself may be downloaded into several nodes of the network.

3) High volume products: these are sold into a very large market-place, typically to very small users, e.g. home computer users, and it may not be feasible to record details of all installations and the amount of use of the product on each. In this case, reliability should be assessed during a trial on a manageable sample of controlled systems (sometimes referred to as a beta test).

4) Mobile installations: where software is installed on a vehicle or on portable equipment, the geographical location may not be applicable, but each set of hardware equipment should still be identified.

7.4.3 *Recording failures*

Users should record perceived failures on an incident report form. This should be referred to as an incident report (rather than failure report) since it may be found on diagnosis that no failure actually occurred and the user was mistaken, or that the incident was actually due to something other than failure of a software item. The term fault report should not be used since the user is reporting an event, and the underlying fault (if any) may only be determined by investigation. The user should provide the following information.

a) Incident identifier: a reference code for the incident unique to the given installation.

b) Installation identifier: so that responses can be sent to the correct place.

c) Originator's name: for receipt of responses and in case further information is requested.

d) Product and version identifier of the software item which is thought to have failed.

e) Date and time of incident, to an appropriate precision.

f) Symptoms observed: a description, and possibly a classification, of the incident.

g) Circumstances: a description of what was occurring at the time of the incident, to assist in identifying the trigger (see **4.3.3**).

h) Severity: a measure of how serious the consequences were for the users. This may determine the priority with which the support team process the report.

The report should be accompanied by any relevant evidence, e.g. memory dumps, screen printouts, file listings, etc. It may be possible to automate the recording of failure in some cases, e.g. where the operating system detects that an application program has issued an illegal instruction and records it automatically in a file. It may be possible to assist the reporter by providing on-line entry into an incident log. Generally the reporting of incidents requires human observation, however.

Following receipt of the report, CDCF should assign an incident identifier unique within the whole organization. After diagnosis, further information should become available. Data that should be recorded for each incident is as follows.

1) Incident identity: code assigned by CDCF, unique within the organization, as well as the code assigned by originator, local to the installation.

2) Product and version: the identity, baseline version, and minor modification level (see **7.1.5**, **7.1.6**) of the software item which failed, as established by diagnosis.

3) Location: identity of installation on which the incident occurred (see **7.4.2**).

4) Time: real time at which the incident occurred. This is needed to identify the first activation of each fault on a sample of installations. Where time to failure data can be collected, e.g. where software is in use on a single installation, amount of system use up to the incident may be entered in the incident record also. Where failure count data is the best available, i.e. in most cases where software is in use on several installations, system use may need to be recorded separately (see **6.4.3.2**).

5) Mode: the set of symptoms which was observed. A textual description of the symptoms should be included. A classification of the symptoms on a nominal scale may be given also, e.g. where a system may output one of a defined set of exception messages. Classification schemes for prominent symptoms such as 'system dead' or 'screen blank' may also be devised.

6) Effect: description and classification of the consequences for the environment in which the system is in use, e.g. 'operating system crash', 'application program aborted', 'slow response', 'loss of data file', 'wrong output', etc.

7) Mechanism: description and classification of the causal chain leading from the initial activation of the fault to the eventual mode of failure observed.

8) Causes: description and classification of the failure causes, i.e. type of trigger, description of trigger, type of fault, and identity of the fault.

9) Severity: measures of the cost of the incident to the user. This may be a classification on an ordinal scale, e.g. critical, major, minor, negligible, or a measure on a ratio scale, e.g. length of time to restore service, or financial loss consequent upon failure.

10) Cost: a measure of the effort and other resources expended by the developer in responding to the incident report.

The above scheme for describing incidents is intended to provide measurements on independent scales. Each item of data may be seen as answering a certain question: 'What failed?', 'Where did it fail?', 'When did it fail?', 'What happened?', 'What were the consequences?', 'How did it happen?', 'Why did it happen?', 'What was the cost to the user?', 'What was the cost to the developer?'.

The reporter of the incident should only be asked to provide information which is known at the time of occurrence: 'Who am I?', 'Where am I?', 'What is the date and time?', 'What has just happened?', 'What was I doing immediately beforehand?', 'What were the consequences?'.

The reporter may be one of the developer's staff where data is collected during trial, or a user where data is collected during live operation. The collection of software failure data should begin as soon as a recognizable system is available. This will generally be after software integration when system testing begins. Reliability assessment using stochastic reliability models should be based on data collected during a period of realistic use. Prior to actual operation, this will involve some form of trial. Failure data collected during this phase can help to indicate when the system is ready for release, and may form part of the evidence gathered from the acceptance test. The procedures for processing incident reports during trial should be the same as those that will be used during operation, so that the trial may be used to validate these procedures as well as to estimate reliability.

The product identity in the failure record should refer to the system level software item (see **7.1.6**). The precise location of the fault within a certain sub-system, module or document should be part of the fault record (see **7.4.4**).

There are four aspects to the cause of an incident.

i) Type of trigger: classified on a nominal scale, e.g. physical hardware failure, operating conditions, malicious action, user error, erroneous report, 'unexplained'.

ii) Description of trigger: a description of the precise circumstances which activated the fault, as reported by the user and confirmed by diagnosis.

iii) Type of fault: classified on a nominal scale, e.g. physical hardware fault, design fault, interface problem, 'no fault found'.

iv) Identity of fault: a coded identifier to serve as a cross-reference to the fault record (see **7.4.4**).

In a complex system, it may be far from obvious at the time of reporting whether an incident is a manifestation of a software fault, hardware design fault, or physical hardware fault. Also, a hardware failure may activate a latent software fault, or vice versa.

Incidents due to malicious actions may also be indistinguishable initially from those due to unintentional causes. Examples are the symptoms of infection by a computer virus, of activation of a 'logic bomb',[1)] or of unauthorized access by an attacker. Such incidents are relevant to security rather than to reliability.

Another class of incident is often diagnosed as 'user error', and the underlying fault as an unfriendly user interface design or deficiency in documentation. Such incidents are relevant to usability rather than to reliability.

Faults may be introduced at any phase of system development. In addition to faults due to mistakes in design or in writing source code, it is important to include problems caused by inadequate definition of system requirements. Although the system behaves according to specification, the user may still have justifiable grounds for complaint.

All of these categories of fault and trigger should therefore be allowed as causes of incident by the recording and classification scheme. There is a danger that important shortcomings of the system may be ignored if the definition of fault is restricted simply to accidental lack of conformance to specification.

In a certain proportion (typically around 10 %) of incidents, even detailed investigaton will fail to diagnose the cause (trigger 'unexplained', and 'no fault found'). It is recommended that all such incidents are treated as manifestations of a fault in design.

Some incidents will turn out to be due to mistaken reporting, but caution should be exercised when consigning a report to the 'erroneous report' category, in case this conceals a genuine usability or requirements problem.

The sets of categories used for detailed classification of mode, effect, cause and severity are specific to the type of system, and it is beyond the scope of this British Standard to provide an all-purpose classification scheme.

Severity (cost to user) should be distinguished from cost of response (cost to developer) since there may be a large disparity between these, e.g. where the user loses large amounts of essential data as a result of a failure, with serious consequential financial loss, but the resolution of the report requires only a few hours of effort from the developer's support team.

In addition to measuring reliability, it may be required to measure recoverability, availability, and maintainability, during trial or in operation.

Recoverability is the ability of the system to resume service following a failure, and may be measured as mean time to restore service (i.e. mean down time) following failure. Availability may be measured as the proportion of time during which the system is able to provide its required service, and is estimated by combining measures of reliability and recoverability. Length of down time should therefore be recorded as a measure of severity for each incident if recoverability and availability are to be measured.

Maintainability is distinct from recoverability where software failure is concerned (see **7.6**). It may be measured as mean cost to respond to an incident report. This is required in order to estimate cost of support of a software item. Cost to developer of responding to the report should therefore be recorded for each incident if maintainability is to be measured.

Incidents in certain categories may be extracted and counted for analysis, e.g. incidents of given mode, effect, or severity, incidents occurring within a given time interval, incidents due to faults in a given software item, etc.

7.4.4 *Recording faults*

Data on faults can only be collected after diagnosis and is therefore recorded by the developer's support team. The following information should be recorded for each fault.

1) Fault identity: a code to enable the fault to be identified uniquely.

2) Location: identifier and version of the software item in which the fault lies (where this can be definitely established: interface and requirements deficiencies may be difficult to locate precisely).

3) Time: a fault is a condition not an event. However the times of three events which define its lifetime may be of interest: time of creation; time of detection; and time of correction. These may be recorded as the phases of development in which they occurred, or as calendar times. The earlier a fault is introduced into the system and the later it is detected, the greater its total cost is likely to be to the developer.

4) Mode: classification and description of what is wrong in the system that constitutes the fault. A fault may be categorized as something missing, incorrect or extra, and assigned a detailed type specific to the type of system, e.g. 'logic error' or 'uninitialized variable', in a scheme consistent with that used to classify 'defects' found in inspection (see **6.2.2**).

5) Effect: the mode of the failure likely to occur if the fault is activated.

[1)] Code designed to cause damage to the system/data on or after a predetermined date.

6) Mechanism: the type of activity which led to the creation, detection, and correction of the fault. A fault may be created by a human mistake during development activities such as requirements definition, design or coding, or during maintenance (either perfective, adaptive, or corrective). It may be detected during inspection, test, trial, or operation. It may be corrected by code change, documentation change, work-around, etc. (see **7.6**). The correction mechanism should include a cross-reference to the relevant change record.

7) Cause: classification and description of the type of human error which led to the inclusion of the fault, e.g. communication error, misunderstanding of application domain, clerical error, etc.

8) Severity: a measure of the severity of the failure which could occur in the worst case if the fault were activated.

9) Cost: a measure of the total cost to the developer of responding to all incidents due to the fault, of diagnosing the fault, and of correcting the fault. This can be extracted from the incident records cross-referred to the fault, and the change records.

7.4.5 *Recording changes*

All modifications to the software should be recorded, and configuration control should be maintained. As part of reliability data collection the following data should be recorded for each change.

a) Change identity: a code to enable the change to be identified uniquely.

b) Location: the identity and version of the software item which is modified.

c) Time: the time at which the change was released to users.

d) Effect: for corrective change. Was the fault rectified? Were any other faults introduced?

e) Cause: reason for the modification identified as corrective (to remove a fault), adaptive (to customize the system to a particular user's requirements or to port it to a new platform) or perfective (to enhance the function of the system).

f) Cost: effort and other resources consumed in devising the change, regression testing, and issuing a corrected version of the system.

7.4.6 *Recording execution time*

It is not possible to apply a stochastic reliability model without a meaningful measure of the extent to which the software has been used and the faults within it exposed to the possibility of activation. This is usually a measure of execution time. The measure chosen should be appropriate to the type of software as follows.

a) Elapsed time (i.e. real time or wall-clock time): This is meaningful only for software which is in use 24 hours a day, seven days a week. For software which is in use for a fixed period every day, e.g. during prime shift, elapsed time may be proportional to the amount of use.

b) System operating time: for software which is executed the whole time that the installation is in operation, e.g. real-time control software, embedded software, or operating system software.

c) Normalized system operating time: where software is in use on several installations which incorporate hardware processors of different speeds, the operating time from each installation may be corrected by a factor to take account of the power of the processor.

d) Program loaded time: for software which is in use in a single programming environment in which it is intermittently loaded, executed, and then deleted.

e) Processor time: for software in a multi-processing environment, where the operating system has the facility to record the amount of processor time consumed by each process.

f) Hands-on time: for interactive software where idle time is of no interest, or where the class of failures being recorded are those due to problems experienced by the user as a result of human–computer interface faults.

g) Transaction count: for interactive software which sends a response to each query from the user.

h) Object instruction count: where a profiling facility is available to record the number of object code instructions executed.

i) Source instruction count: where software instrumentation in the form of recording probes compiled into the software at various points has been implemented, or where the software is run on a test-bed which records the number of source instructions executed.

j) Number of demands: for 'one-shot' software which is called upon periodically to perform a specific task, e.g. safety protection system software monitoring an industrial process.

Where a software item is in use on several installations, measurements of its use should be collected from all installations in the sample being studied and then combined to give the total use in each defined period, e.g. the total use may be recorded week by week, or day by day, depending on the precision required. This will yield failure count data (see **6.4.3.2**).

Automated recording is desirable to ensure that records are complete. Where structural models are being applied, automated recording using software instrumentation or similar is required in order to measure an execution profile (see **6.4.21**).

7.4.7 *Software reliability data storage and extraction*

7.4.7.1 *Use of corporate database*

It is strongly recommended that all software reliability data that are collected are held in an electronic database for ease of extraction and analysis. This should be done preferably on a company-wide basis so that information about different development projects and their delivered software products can be compared easily. Such a database is an important part of corporate memory.

The reliability modelling activity requires the accumulation of records over long periods of time, longer generally than the life of an individual item.

The database should be administered by a CDCF so that data integrity and comparability across projects can be maintained. Past projects to set up data repositories to which many companies or other organizations could contribute have experienced difficulties due to the different practices of data collection in use in different companies.

The database may contain any or all of the following types of data described in this British Standard.

a) process data, e.g. cost, duration, quality of personnel, development practices used;

b) product properties, e.g. size, complexity, structure, design type;

c) failure data: software items, installations, incidents, faults, changes, execution time.

There are several uses for this data, including assessment of achieved software reliability, management of software support, improving the development process by modelling the effects of different development processes on achieved software reliability, control of current projects using measured values of indicators, and making estimates for future projects based on historical data (see **5.2**).

Reliability models based on development process attributes (see **6.2**) or software properties (see **6.3**) evaluate indicator measures (see **4.4**) whose correlation with achieved reliability in operation should be established independently. A corporate database including all types of data (process, properties, and failure) is a prerequisite for the development of such models.

The following clauses describe recommended data structures for storage of raw software failure data in two simple cases, and procedures for the extraction of time to failure and failure count data (see **6.4.3.2**).

7.4.7.2 *Relational database concepts*

The recommended structures are defined using the relational approach. A relational database consists of several named tables, each of which contains data about an entity type, e.g. incident, fault, etc. A table is referred to as a relation, hence the term relational database.

A table has a fixed number of columns, each of which represents an attribute of the entity type, e.g. incident identifier and location are attributes of the entity type incident, and are represented by columns in the incident table. Any attribute takes values of a fixed data type, e.g. integer, character, decimal, etc.

An occurrence of an entity, e.g. a record of a particular incident, is represented by entering a data value for each attribute under the appropriate column in a single row of the table. Where the value of an attribute is unknown or inapplicable, a null value is entered. Each table has a primary key, which is a set of attributes whose combined value is unique within all the rows of a table, e.g. incident identifier is the primary key of incident. Since the value of the primary key is used to refer unambiguously to the rows in a table, no attribute in a primary key may contain a null value.

Any piece of data in the whole database, i.e. the value of any attribute of any occurrence of any entity can then be accessed by giving the table name, column name, and the value of the primary key.

The structure of a relational database consists of several many-to-one mappings between pairs of tables. Each mapping is established by a foreign key, which is a set of attributes in one table whose combined value is required to match the value of the primary key in one row of another, so acting as a cross-reference between the two tables, e.g. fault identity is a foreign key from the incident table to the fault table, representing the fact that several incidents may be due to activations of a single fault.

Mappings may be into or onto the target table. If table A is mapped into table B, then with each row in A one row in B is associated, but there may be rows in B which are not associated with any row in A. If in addition every row in B is associated with at least one row in A, then the mapping is said to be from A onto B.

A foreign key may be allowed to take a null value in some cases, e.g. before the cause of an incident has been diagnosed, it may be recorded in the incident table with a null value in the fault identity column.

To represent a many-to-many relationship between the rows of two tables, a linking table is inserted between them. This consists solely of pairs of primary key values from the two related tables.

Foreign keys provide cross-references which are followed during data extraction, and facilitate consistency checks to preserve the integrity of the data structure during insertion, amendment, and deletion of data. The relationships between the tables represent relationships between the corresponding real-world entities.

Structures such as these may be implemented by using any proprietary relational database management systems (RDBMS).

The database structures described here are illustrated by implication diagrams, in which mappings into are denoted by single arrows between tables, and mappings onto by double arrows, as illustrated in figure 12.

Figure 14 illustrates tables and attributes for single installation data. The different types of table attribute are denoted by different formatting as follows.

TABLE (**primary_key**: *foreign_key*; other attributes)

Where an attribute is both a primary key and a foreign key, it is displayed in ***bold italic***.

Figure 13 illustrates a simple structure, from which time to failure data may be extracted (see figure 4), and figure 15 illustrates a simple structure from which failure count data may be extracted (see figure 5).

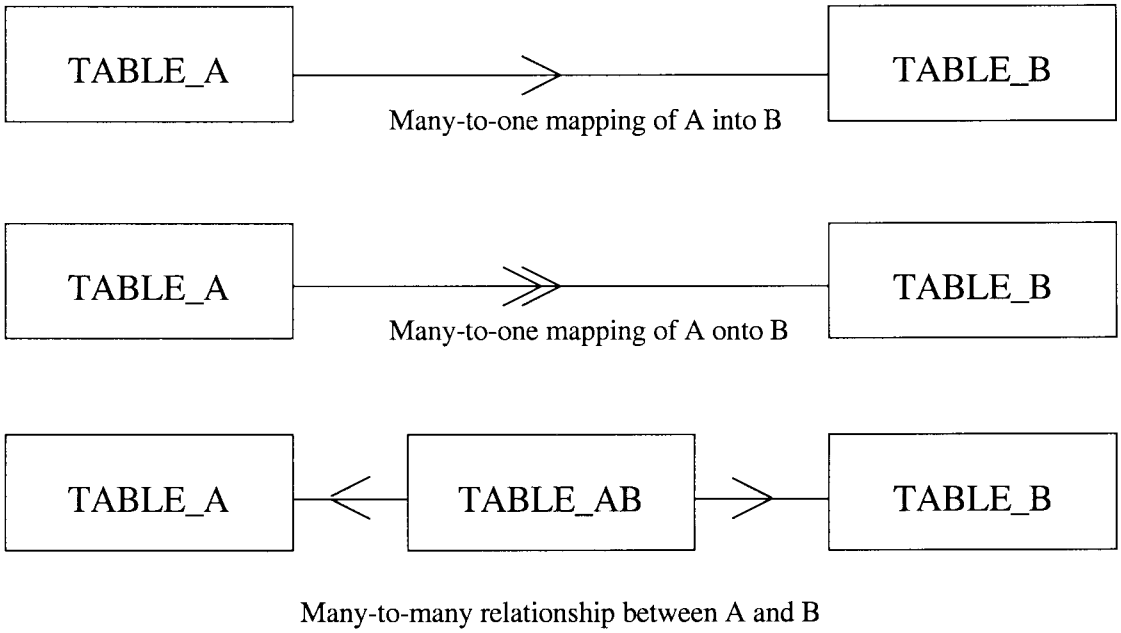


Figure 12. Graphical notation for relationship database structure

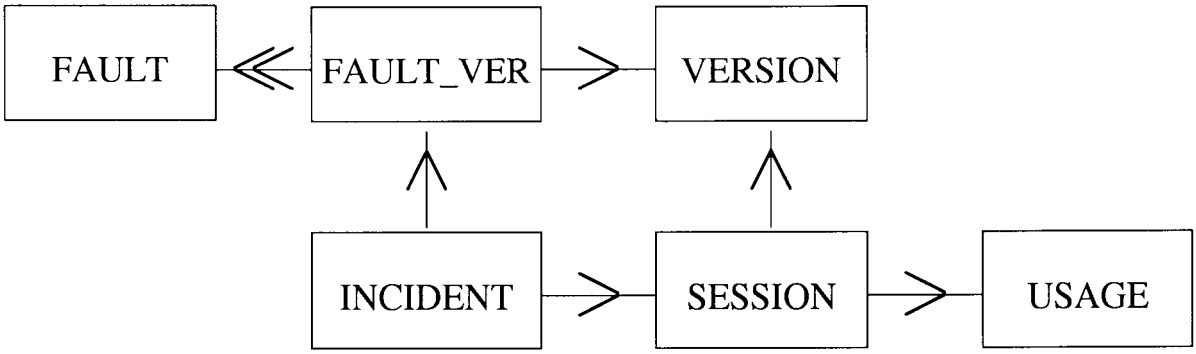


Figure 13. Database structure: single product on single installation

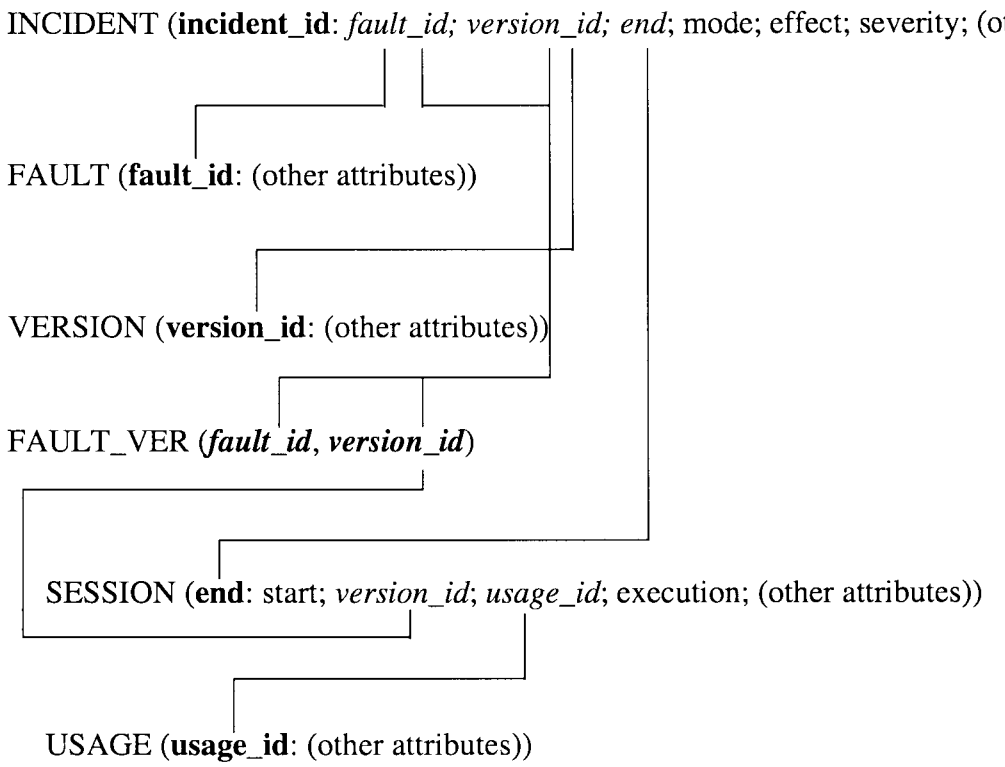
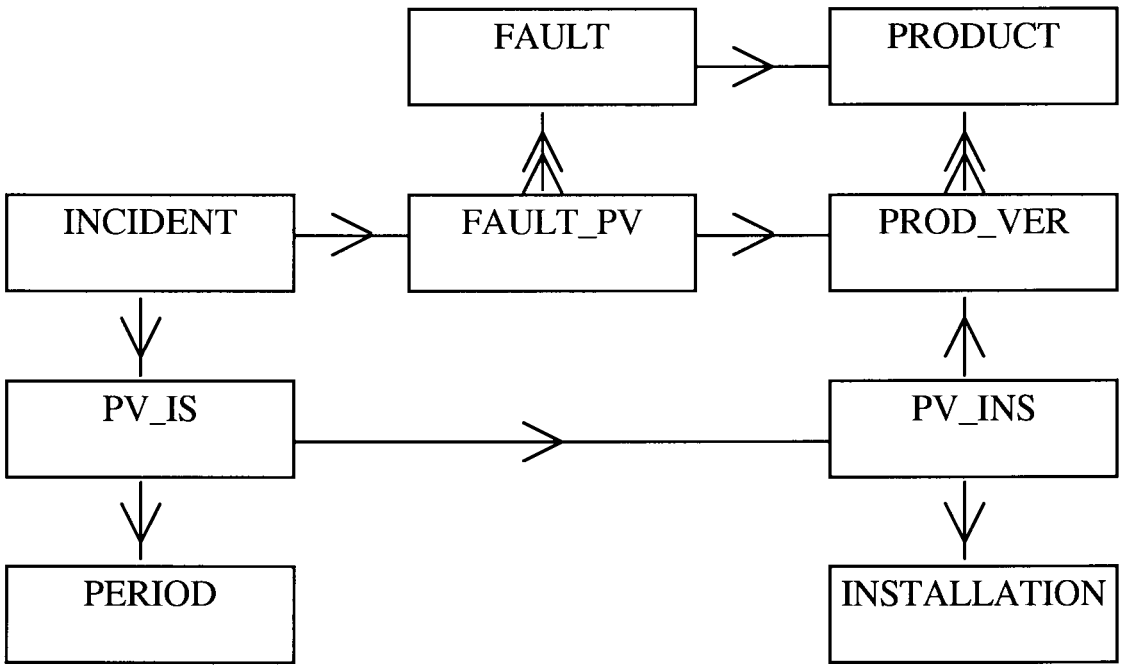


Figure 14. Tables and attributes for single installation data



Key to abbreviations:

INS	installation
IS	installation session
PROD	product
PV	product version
VER	version

Figure 15. Database structure: multiple products on several installations

7.4.7.3 *Data structure for single installation*

Figure 13 illustrates the structure of a database intended to hold the raw failure data collected from a single software product executed on a single installation. Figure 14 illustrates the tables and attributes. Only attributes used in data extraction are shown, although other descriptive attributes may be included.

The software may pass through several baseline versions in the course of trial or operation. Its operation consists of a series of sessions. In each session, only one version of the product is used, and is subjected to one particular usage, or operational profile. A session is deemed to be terminated either by a shift to another usage or by an incident. Sessions do not overlap, and therefore a session can be uniquely identified by its date and time of termination, referred to as **end**, which is the primary key of SESSION. Each incident is therefore cross-referred to one session (that which it terminates) by the foreign key *end* (which is also the record of when it occurred), and each session is cross-referred to one version and to one usage by the foreign keys *version_id* and *usage_id*. (It is assumed that each usage can be identified by a simple code.)

Execution time in each session (not necessarily the same as the elapsed time of the session) is recorded in the attribute SESSION.execution.

NOTE. 'A.b' denotes 'attribute b of table A'.)

Each fault may be present in more than one version of the software. There is therefore a many-to-many relationship between FAULT and VERSION, via the linking table FAULT_VER. An incident is due to the activation of a fault in a particular version, and so INCIDENT is cross-referred to FAULT_VER by the multiple foreign key (*fault_id*, *version_id*). Each of *fault_id* and *version_id* may also be considered as separate foreign keys to FAULT and VERSION.

Figure 14 illustrates the connections between the keys.

7.4.7.4 *Extraction of time to failure data*

The procedure for extracting time to failure data for a given version of the software under a given usage is as follows.

a) From INCIDENT select all incidents for which the record in SESSION whose key is INCIDENT.*session_id* contains values in SESSION.*version_id* and SESSION.*usage_id* which match the version and usage chosen for analysis.

b) Group the resulting subset of incidents by INCIDENT.*fault_id* and sort by INCIDENT.*end*, to give groups of incidents, each group due to activations of the same fault and ordered by time of occurrence.

c) Select the first incident in each group, to give a list of first activations of each fault in the given version under the given usage, ordered by time of occurrence.

d) For each incident in this list, sum SESSION.execution where SESSION.*version_id* matches the version, SESSION.*usage_id* matches the usage chosen for analysis and SESSION.end is less than or equal to INCIDENT.*end*.

The result is a list of execution times from start of operation to first activation of each fault. This can be converted to a list of inter-failure times by taking the difference between successive entries.

7.4.7.5 *Data structure for multiple installations*

Figure 15 illustrates a database structure to hold software failure data collected from a number of separate software products, any version of any one of which may be in operation on any one of several installations. This situation is frequently encountered by developers of commercial software who sell a range of products into a large marketplace. Failure count data (see **6.4.3.2**) is the best that can be collected in such cases.

Figure 16 shows the tables and attributes of this database, and the foreign keys which establish the relationships between the tables. As in the previous example, only attributes which establish this structure, or are used in the example of data extraction which follows (see **7.4.7.6**) are shown. Other attributes to contain other data contained in incident reports (see **7.4.3**) and fault records (see **7.4.4**) will normally be included.

PRODUCT is a table of the products whose reliability is to be assessed, and PROD_VER is a table of all the extant baseline versions of these products.

INSTALLATION is a table of all the installations on which the products may be executed, and PV_INS records that a given baseline version of a product is present on a given installation between given dates of delivery and withdrawal (PV_INS.deliver, PV_INS.withdraw).

PERIOD contains the start and end dates (PERIOD.startp, PERIOD.startp) of each of several successive periods of calendar time. Failure counts and execution time are accumulated period by period, and it is assumed in this example that these periods are the same for all products. (This constraint could be relaxed, however.)

PV_IS contains the start and end calendar times (PV_IS.starts, PV_IS.ends) of each session of use of each version of each product on each installation, with the amount of execution time (PV_IS.execution) measured during the session. Each session is assumed to lie wholly within a single period.

FAULT contains a record of each fault that has been detected in each product. Since a fault may be present in more than one baseline version of a product, there is a many-to-many relationship between FAULT and PROD_VER, via the linking table FAULT_PV.

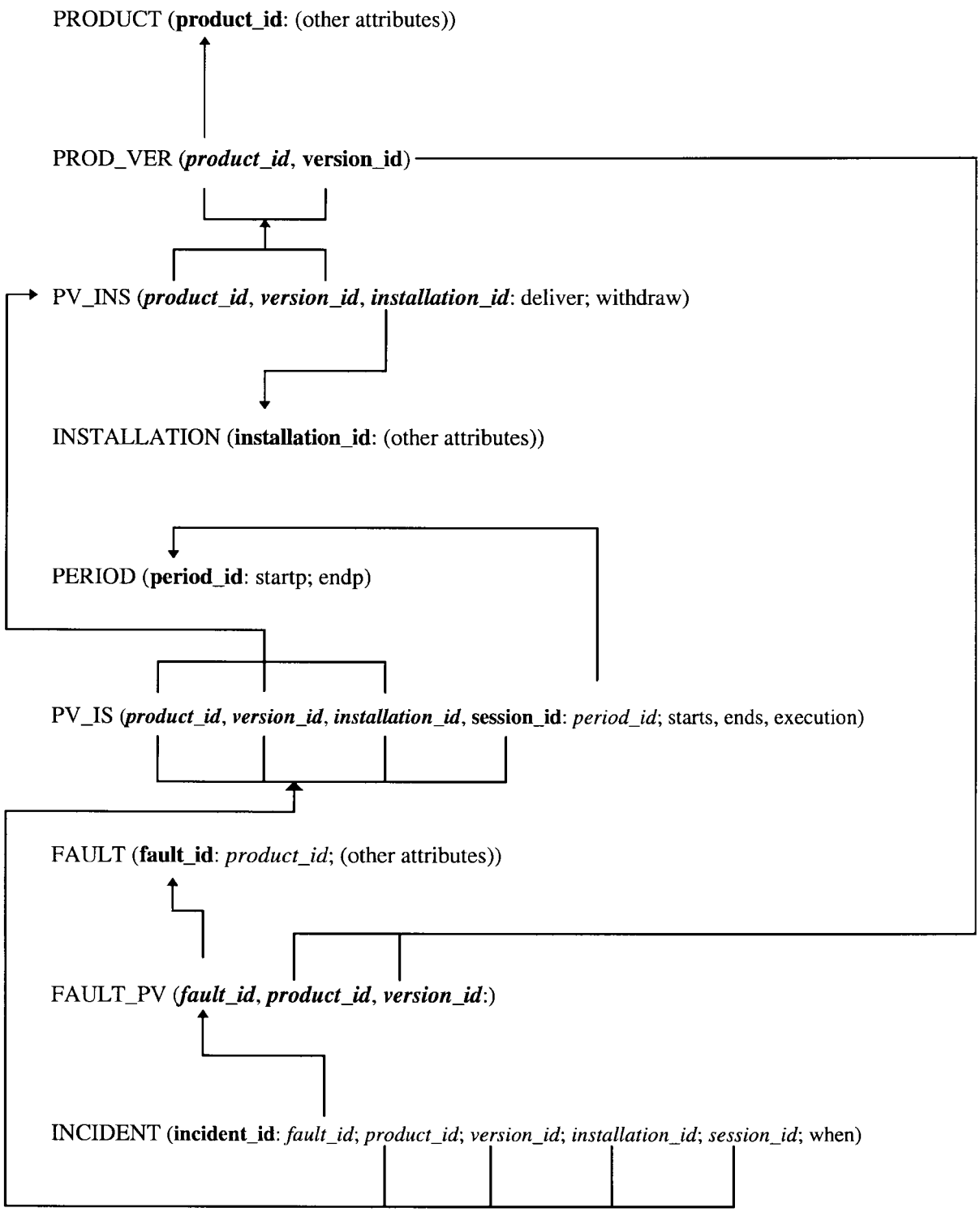


Figure 16. Tables and attributes: multiple products and installations

INCIDENT is a table of each incident recorded during the use of any of the products, with a record of its calendar date and time of occurrence (INCIDENT.when). Its relationship to execution time is established by the fact that it contains a foreign key into PV_IS.

7.4.7.6 *Extraction of failure count data*

The procedure for extracting a set of failure count data which can be analysed to measure the reliability of a single baseline version of a single product over all installations is as follows.

a) From INCIDENT select records for which INCIDENT.*product_id* and INCIDENT.*version_id* correspond to the product baseline chosen for study, to give a table of relevant incidents.

b) Group the relevant incidents by INCIDENT.*fault_id* and sort them within each group by INCIDENT.when.

c) Select the first incident from each group, to give a table of relevant first activations of each fault.

d) Count the first activations in each period as determined by INCIDENT.when lying between PERIOD.startp and PERIOD.endp.

e) Sum PV_IS.execution where PV_IS.*product_id* and PV_IS.*version_id* match the product baseline chosen for study, grouping the records for summation according to PV_IS.*period_id*.

The result is a list of pairs of numbers, the total execution time and count of new faults detected in each of the successive periods.

7.4.7.7 *Enhancements to basic structures*

Simple examples have been given for purposes of illustration. They may be extended as follows.

a) Include records of system usage for each product in the multiple installation structure.

b) Record changes as well as incidents and faults.

c) Include records of down-time and maintenance effort, and extract data for the assessment of recoverability, maintainability, and availability.

d) Incorporate facilities to manage the handling of incident reports, including recording report status, issue of responses, etc.

e) Include full software item data, breaking down the software structure into sub-systems and modules, and cross-referring faults to the modules in which they are located.

f) Additionally record execution profile in the structured case to extract data for use with structural models.

Any actual data collection exercise is likely to pose its own specific problems, and require the adaptation of the basic approach recommended.

7.5 Data collection forms

7.5.1 *General guidance on forms*

Forms used for data collection should be simple to use and collect only that data which an organization has decided is needed for set purposes. This clause provides guidance on data collection by illustrating how basic information may be captured. Extra information which may be necessary depending on the method and models to be used is described but not shown on the example forms.

The three basic forms described in this clause are intended to be used in collecting data concerning the occurrence of incidents and the amount of software use (typically measured by execution time). Such data are needed for many types of model, e.g. for assessment of reliability, for estimating project support costs, and for analysis of the causes, methods of detection, categories, consequences and sources of faults (see **5.2**).

7.5.2 *Form 1: incident report*

The term incident denotes an unforeseen event occurring during the test phase or during normal operation. An incident may or may not amount to a failure (see **7.4.2**).

This form normally comprises two parts (see figure A.1). The first part is to be completed by the user of the system and the second by the producers or maintainers of the system, or by the CDCF (see **7.1.3**). The first part of the form contains the following information.

a) Organization. The name of the organization which is reporting the incident.

b) Incident identifier. A code to identify the incident uniquely within the installation.

c) Date and time of incident.

d) Installation identifier and configuration. The name or code by which the installation is formally identified together with its configuration (if known).

e) Version. The revision level of the installation platform.

f) Report completed by. The name of the person who has completed the incident report.

g) Date report completed. The date the report was completed which may not be the same as the date of the incident.

h) Description of incident. A description of the incident as perceived by the user of the system.

i) Perceived impact/severity. The effect that the incident has had on the user. Where possible the severity should be classified into different categories such as minor, major, severe.

The second part of the form contains:

1) Responder. Name of the person who replies to the incident.

2) Date of response.

3) Action to be taken. This should contain details of activities necessary to avoid a recurrence of a similar incident. In cases where it has been established that no failure has occurred, this may simply be to correct a user misunderstanding.

4) Status (open/closed). The status of the incident report should be stated for the information of the originator. 'Open' and 'closed' are a suggested minimum set. Many organizations may have a wider set or may use different categories depending on their needs.

The above items are a suggested minimum set. Depending on the organization's needs other items as described earlier such as cross-references to other reports, location and source of fault will need to be recorded (see **7.4.2** to **7.4.4**).

7.5.3 *Form 2: software item use log (calendar time)*

Systems containing software may employ many different types of interface with the user. The software use log (calendar time), is used to record the amount and nature of use (see figure A.2).

a) Life cycle phase. The phase of the software life cycle to which the use applies. Examples of such phases are: unit test; integration test; system test; and system trial. The precise description used will depend on the particular software life cycle definition employed.

b) System/installation. This is the identity of the specific equipment in use.

c) Date/time. The start time of each session of a particular mode of use should be recorded.

d) Mode of use. This should be taken from a list of agreed codes or abbreviations representing all the possible modes of use. These should include 'power on', 'power off' and 'idle', so that only the start of each session needs to be recorded. (The end of a session is the start of the next, which may be a session of 'idle' time.) The modes of use will in many cases be the name of the particular item of software being run. Many different items may therefore have their time recorded on a single form, but if necessary, a different set of forms may be used for each item;

e) Incident identifier. When an incident occurs, the time should be recorded together with the incident number, which is a cross-reference to the associated incident report. Any incident is deemed to terminate the session in which it occurs. This may be followed by a session of 'recovery'. When work is resumed, the start of a new session is entered.

f) Comment. This should record any relevant observations on the session or incident.

7.5.4 *Form 3: software item use log (use time)*

This is intended to record the amount of use of a single item at a single installation in each of a succession of pre-defined calendar time periods. Together with the incident reports, it allows the recording of failure count data (see **6.4.3.2**). It should contain the following.

a) Life cycle phase. The phase of the software life cycle to which the use applies. The precise set of phases used will obviously depend on the definition of the software life cycle in use within the organization.

b) Units of use. The chosen measure of use, for example, the execution time in CPU seconds, or a count of transactions processed by an interactive program. The units of measurement chosen should be specified.

NOTE. A suitable means of recording should be available, for example, automatic monitoring by the operating system of CPU time consumed by each process, or instrumentation code to maintain a count of transactions.

c) System/installation. This is the identity of the particular equipment in use.

d) Software item/version. This is the name of the item and its version.

e) Period (length). The period is some appropriate length of calendar time. It might be, for example, a day, week, or month.

f) Period (identification). A period may be identified in any convenient way, for example date of week ending (or the international week number), the organization's accounting period number, etc. If the item is in use on several installations, the periods should be identical for analysis to be possible. The length of the period, and its means of identification, should be recorded.

g) Period versus use. The period identifier and amount of use are recorded in pairs in the columns, reading from top to bottom and left to right. No cross-references to incidents are made on this form. Incidents can be allocated to period by matching the date and time on the incident report with the pre-defined start and end date of each period.

7.6 Logistics of software maintenance

7.6.1 *Overview of software support*

Following the release of a software product to the customer(s) for use, support is normally provided. The main objective of software support is to gain marketing advantage by increasing customer satisfaction. It may include answering queries, providing advice and training, responding to incident reports, and progressively improving the product. Improvement requires modification of software in the field, usually referred to as maintenance, and poses logistical problems whose management requires an assessment of software reliability. There are three types of maintenance, performed for different reasons.

a) Corrective. Removal of faults after release. These may be detected either by diagnosis of the causes of incidents reported by users, or by other means, e.g. re-inspection of source code. Corrective maintenance does not normally require a change of baseline. Reliability growth is usually observed as faults are removed.

b) Adaptive. 'Customizing' the product to the requirements of a particular user or group of users, or adapting it to cope with a new environment, e.g. a new hardware/software platform. Adaptive maintenance may not require a change of baseline, but leads to different modification levels on different installations, needing careful configuration management. Adaptation may also cause an initial decrease in reliability.

c) Perfective. Enhancement of the product to perform new functions that were not stated in the original requirement, in order to meet changes in the needs of the customer(s) following experience with the previous version, or to gain a marketing advantage by improving the product. Perfective maintenance usually requires a change of baseline, and the new version may be treated as a new product for purposes of reliability assessment. Initially the enhanced software may be less reliable than the original version due to the inclusion of new code containing new faults.

Management of software support involves several activities.

1) Support cost estimation prior to release. This will affect commercial decisions such as setting the price of the product and of any maintenance contract, whether to offer a warranty, etc.

2) Assessment of customer satisfaction in terms of the estimated cost of ownership to the customer, reliability and availability as perceived by the users on each installation, and whether support is seen to be adequate.

3) Choice of methods and organization of support, including the number of staff to be employed, how to organize the support teams, which techniques to use, and when to release new versions of software, with the aim of maximizing customer satisfaction while minimizing support cost.

4) Forecasting. 'What if...?' exercises to forecast the effects on support cost and profits of growth or shrinkage of the market, of growth or decay in the reliability of the product, of variations in the size and organization of the support teams, etc.

Measurement of software reliability (and also recoverability, maintainability and availability) is necessary for all of these.

It has been found that up to 70 % of the cost of producing software is expended on maintenance.

7.6.2 *Logistical considerations*

7.6.2.1 *Factors affecting software support*

Maintenance of hardware requires the equipment to be brought to the support personnel or the personnel to visit the installation. Transport and storage of spare parts are major logistical considerations.

Maintenance of software generally requires only the movement of information. 'Evidence' (memory dumps, etc.) is sent to the support teams with every incident report. Modifications (software 'patches', new software releases, 'work-around' procedures, etc.) are sent to the customer in response. Recording failures, faults and changes, and configuration management, are the main logistical problems.

Four factors that affect the software support operation should be considered.

a) Software reliability. The reliability of the product (and also its recoverability, maintainability and availability), and whether reliability growth under corrective maintenance is rapid or gradual.

b) Field characteristics. Number of installations, and the rate of increase or decrease, types of installation and how the environment affects the perceived reliability of the software product on each type of installation.

c) Support organization. Incident reporting and response procedures, number and size of support echelons, tendency for queues to build up during the processing of incident reports, etc..

d) Maintenance techniques. Use of 'fix-on-fail', fault clearance release, and publication of known faults. These techniques differ in their effectiveness, and the timing of certain actions may be important.

The effects of each of these factors are considered in more detail in **7.6.2.2** to **7.6.2.5**.

7.6.2.2 *Effect of reliability*

The rate at which incident reports are received is driven by the perceived reliability of the software on customer installations. This may be seen as the interaction between an 'inherent' reliability and environmental factors specific to the various installations.

The inherent reliability should be measured using the methods of **6.1**, and should include an assessment based on records of failure and execution time collected during a trial under realistic conditions. This assessment may be used to predict the failure rate of the system in operation.

Allowance should be made for the fact that the failure rate in operation is often found to be higher than that observed in trial, since the trial conditions may be an imperfect representation of conditions in the field.

Black-box software reliability growth models can also predict the change in the failure rate as faults continue to be removed after release. The change will be gradual if the product contains a large number of small faults (i.e. with small individual activation rates), but more rapid if it contains relatively few large faults, since in the former case, the removal of each fault has only a slight effect on the total failure rate.

During trial, the effort to diagnose and correct each fault should be recorded, since this is the other main cost driver for corrective maintenance. The recovery time of the system following each failure should also be recorded, so that the impact on the customer may be predicted.

The failure rate at release, the degree of change expected, the expected recovery time, and the expected cost of diagnosis and correction of each fault may be used to predict the cost of corrective maintenance. These predictions should be validated by continued measurement during operation.

7.6.2.3 *Effect of field characteristics*

The cost to the developer of performing corrective maintenance depends on the rate of receipt of incident reports from the whole field. This in turn depends on the number of installations, which may be predicted from expected sales figures.

It has been observed that the rates of submission of incident reports relating to the same product may differ by factors up to 10 between different installations. The causes of such variation are sometimes obscure, but factors that affect it may include the following.

a) Workload. The number of users on-line to the installation, number of application processes being run, total throughput of work, etc., constitute a 'stress factor'.

b) Usage. The way in which the product is used varies between installations, e.g. on one installation the customer may be developing software in-house whereas on another the customer may be running only off-the-shelf applications.

c) Responsiveness. On some installations, users may be more willing to report incidents than on others, e.g. pressure of work may mean that there is little time to complete report forms, or a high degree of concern about reliability may mean that users are unusually conscientious about making reports.

Such variations are difficult to predict prior to release. It is recommended that data is collected during operation so that reporting rates from individual installations may be compared and these empirical observations used to refine predictions.

7.6.2.4 *Effect of support organization*

A developer's support organization usually consists of a number of echelons, i.e. teams organized hierarchically, each with a defined function. The following levels are commonly found.

a) Service desk. This is a point of contact for customer queries. It performs an initial scan of incident reports and responds to those which can be answered immediately.

b) Support centre. Incident reports which require deeper investigation are passed to a team of specialists in fault diagnosis and removal.

c) Design authority. Difficult problems which cannot be solved at the preceding levels are investigated by the development staff responsible for the original design of the product.

The flow of incident reports between the echelons should be controlled by the CDCF (see **7.1.3**).

The effectiveness of the echelons will depend on the maintenance techniques in use. Typically the lowest echelon will respond to reports of incidents due to repeated manifestations of faults which have been detected previously. The symptoms observed when a fault is activated should be recorded on the company database to assist in the recognition of any future activation.

A report of an incident due to a hitherto undiscovered fault will be passed up to the higher echelons, and may cost 100 times as much to process as a report of a repeated manifestation of a known fault. When supporting large system software in the support centre echelon, a specialist may be expected to deal with around three incident reports per week.

Queues of reports building up in the various echelons will affect the speed of response, and hence customer satisfaction and the improvement in product reliability in operation. The processing of the workload will depend on the level of priority assigned to each report. This should be assigned by the CDCF based on the severity of the incident as perceived by the customer but revised to ensure that a uniform classification of priority is applied to the whole field.

Commercial decisions regarding the terms on which support is offered will also affect the cost to the developer of providing a service. Examples of such commercial decisions are given below.

1) Free service. All incidents reported are investigated and any faults discovered are removed without charge. This has the advantage of maximizing the level of reporting and hence the quality of data collected and the growth in product reliability in operation. The disadvantage is that a high failure rate may result in the developer making a loss on the product.

2) Warranty period. Incident reports are investigated free of charge for a defined period following delivery. The advantage is that the customer gets a free service for that period. The disadvantage is that data collection and product improvement are cut off afterwards.

3) Maintenance contract. For a defined recurrent charge all incident reports are investigated. The advantage is that the developer covers the cost of the service. Disadvantages are that customers may be deterred by the charge and that data collection and product improvement are restricted to those customers who decide to take up the maintenance option. The cost of the contract should be determined to be acceptable to customers and to cover the amount of support activity expected. A maintenance contract may be offered by the original supplier or by a third party maintenance organization.

4) Charge per call. The customer pays a handling charge for every incident report or enquiry. Again the developer covers the cost of the service. Reporting of incidents may be biased towards those with higher levels of severity.

5) No service. For high volume off-the-shelf products no maintenance is usually offered, except that customers may report incidents, and faults diagnosed may be made available in the form of a public list of 'declared deficiencies'. Fault removal may be done when the next release is made available, and customers are required to buy this as if buying a new product. Reliability assessment in operation under these conditions is a hit-or-miss affair.

7.6.2.5 *Effect of maintenance techniques*

Several techniques of providing corrective maintenance are commonly used and have different degrees of effectiveness as follows.

a) Fix-on-fail. After investigation of a reported incident a response is sent to the customer describing the fault which was diagnosed (if any) and the modification required to remove the fault (or in some cases a work-around procedure) and prevent a repeated activation. (With many software products the customer has the facility to apply a patch or modification on site.) Advantages are that the customer is kept satisfied by the rapid response. Disadvantages are that this practice only removes one instance of the fault from a single installation, and so has a very slight effect on the flow of incident reports into the support organization from the whole field, particularly where the product is in use on many installations. In addition, this practice rapidly leads to individual installations being at different modification levels and poses configuration management problems.

b) Known faults database. The corporate database (see **7.4.7**) should include a record of every fault detected in every product. To this should be cross-referred the records of all incidents diagnosed as being due to the fault being activated, and records of any modification devised to remove the fault. This information may be made available to customers. Advantages are the following:

1) following an incident, the customer may compare the circumstances in which it occurred and the mode of failure observed with the trigger and expected symptoms of activation recorded against each fault, and so diagnose the fault responsible without recourse to the support teams;

2) the customer may also follow the cross-reference from the fault record to the record of the modification which removes the fault and apply this modification on site;

3) the customer may apply corrective modifications before observing any incident due to the relevant faults, as a form of preventive maintenance;

4) this improves product reliability as perceived by all customers and reduces the rate at which the support organization receives incident reports from the whole field. It is a far more effective strategy than fix-on-fail.

Disadvantages are the following:

i) incidents other than those due to the activation of a hitherto undiscovered fault tend not to be reported. Most black-box software reliability growth models require only records of first activation of each fault, however;

ii) issues of confidentiality may arise if customers have access to each others' incident reports. This may be avoided by appropriate access restrictions.

c) Fault clearance release. A new baseline version of the product from which all known faults have been removed is released to the whole field. This has a dramatic effect both in improving customer perceived reliability and on the flow of incident reports to the support organization.

The timing of such a release is important. Sufficient time should be allowed for the number of faults discovered in the field to accumulate, otherwise the release will not be as effective as it might otherwise be. If it is delayed too long, then customer-perceived reliability and support team workload will suffer. A new release to provide functional enhancement will have the opposite effect, since new faults will be delivered with the new code.

7.6.3 *Interaction of support cost drivers*

Figure 17 illustrates the interaction of the various drivers of support cost. Sales revenue and maintenance charge drive up both the developer's profit and the customer's cost of ownership. Market growth tends to increase the various levels of 'stress' to which the product is exposed, and this, combined with the inherent failure rate, drives up the perceived failure rate in the field. The queue of incident reports in the support organization increases with the number of installations (which in turn is driven by market growth) and the perceived failure rate. This increases support cost, but the effect on the developer's profits will depend on the terms of support.

The larger the queues of incident reports, the longer the average time that will elapse before any given report is investigated and a response made. This delay will make both the known faults database and any fault clearance release less effective and lead to further reports of incidents due to repeated activation of faults. A vicious circle of increasing queues and increasing delays will exist, and may be exacerbated by product enhancement, which tends to drive up the inherent failure rate.

Successful management of software support depends upon having adequate support effort available to cope with the expected workload (particularly just after release when workload is likely to be greatest) and so avoid such a vicious circle developing.

Queuing theory may provide a means of predicting the behaviour of a support organization, but in such cases (multiple servers, multiple queues, prioritization and non-constant arrival times) a closed-form solution is unlikely to be possible. The use of simulation may be considered.

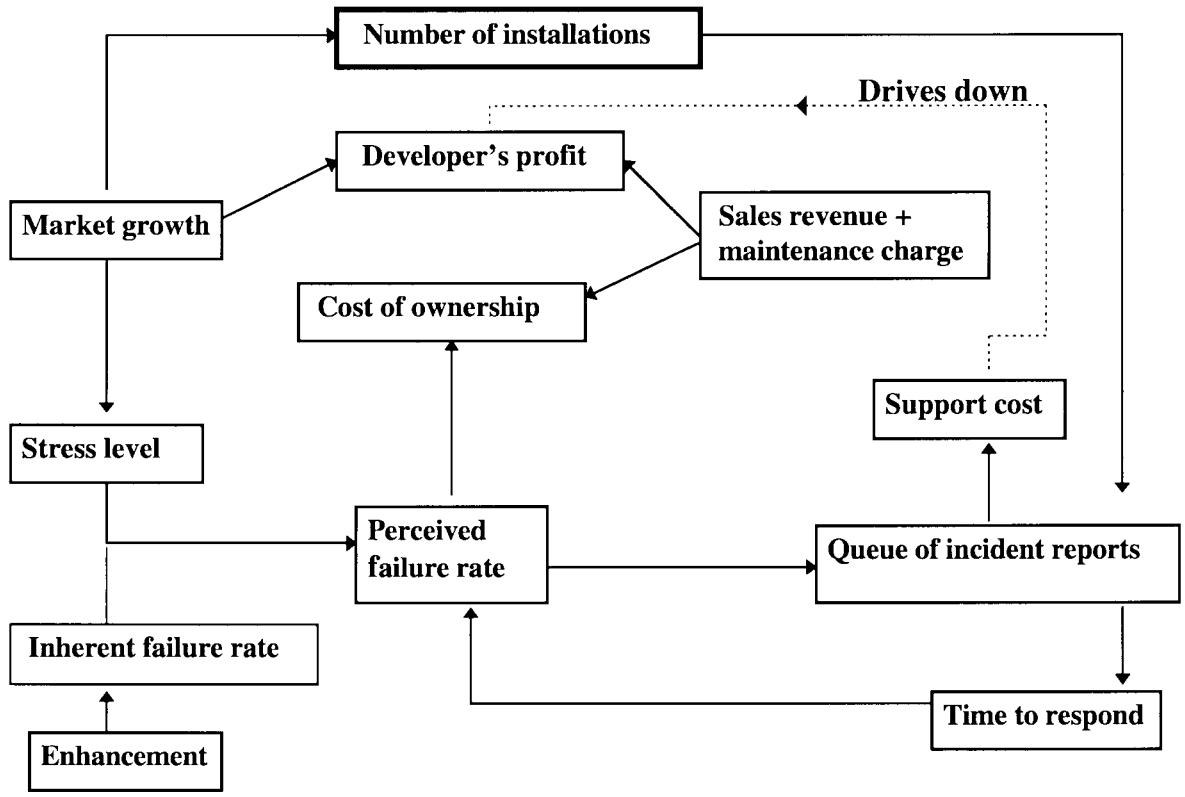


Figure 17. Interaction of support cost drivers

Annex A (informative)
Forms used in data collection

Examples of forms used in data collection are shown in figures A.1 to A.3

Organization.....

Incident identifier..... Date and time of incident.....

System identifier and configuration..... Version.....

Report completed by..... Date report completed.....

Description of incident.....

.....

.....

.....

Perceived impact/severity.....

.....

.....

.....

.....

.....

.....

For office use only

Responder..... Date of response.....

Action to be taken.....

.....

.....

.....

.....

.....

Current status (open/closed).....

Figure A.1 Form 1: incident report

Life cycle phase		Page no. ..... System/Installation ..... Device .....
Date/time	Mode of use/incident identifier	Comment

Figure A.2 Form 2: software item use log (calendar time)

Page no. .....

Life cycle phase ..... Units of use..... System/Installation .....

Period..... Period ..... Software
(length) (identification) item/version .....

Period	Use	Period	Use	Period	Use	Period	Use	Period	Use

Figure A.3 Form 3: software item use log (usage time)

Annex B (informative)
Mathematical descriptions of stochastic reliability models

B.1 Introduction

This annex contains mathematical descriptions of some of the software reliability models introduced in **6.4**. It is intended to assist users of the standard to apply the models using custom-written or off-the-shelf' statistical software. The descriptions are terse, since they supplement the more discursive descriptions in the body of the standard. Important formulae are quoted, but proofs are not included. Its structure follows the classification of stochastic reliability models illustrated in figure 3, although not all classes of models are covered.

B.2 Definitions of terms

NOTE. The acronyms and mathematical terms defined below are used in both annex B and annex C.

B.2.1 *Abbreviations*

CDF	Cumulative distribution function
DU	Duane model
(E)OS	(Exponential) order statistic
G-O	Goel-Okumoto model
IID	Independent identically distributed
J-M	Jelinski-Moranda model
L-V	Littlewood-Verrall model
LCM	Least concave majorant
(L)LF	(Log) likelihood function
(L)LFC	(Log) likelihood function (failure count data)
(L)LFT	(Log) likelihood function (time to failure data)
LSD	Least squared distance
LSE	Least squares estimation
LSRG	Littlewood stochastic reliability growth model
MeTTF	Median time to failure
MLE	Maximum likelihood estimation
M-O	Musa-Okumoto model
MTBF	Mean time between failures
MTTF	Mean time to failure: $E\{T\}$
MTTR	Mean time to repair (i.e. to remove fault)
MTTRS	Mean time to restore service
(N)HPP	(Non-) homogeneous Poisson process
PDF	Probability density function
P-G	Poisson-gamma model
PL(R)	Prequential likelihood (ratio)
ROCOF	Rate of occurrence of failure
RV	Random variable

B.2.2 *Mathematical notation*

Random variables are denoted by upper case letters, and their realizations by the corresponding lower case letters. Both are given below where some models treat a given quantity as a random variable, while others treat it as a fixed but unknown quantity.

$A(x|b)$ denotes the value of a function $A(x)$ of x conditional on event b, or on RV B having realized value b. A_i denotes the value of variable or function A at the end of the ith period or at the ith failure, or for the ith fault, depending on the context.

The use of the variables t, x, k, c, and u, and the corresponding RVs, in describing the process of failure, is illustrated in the body of the standard. Note the distinction between 'time to failure' data (figure 4) and 'failure count' data (figure 5). Unless otherwise stated, it will be assumed in annex B that each 'failure' is the first activation of a fault, and that 'time' is a measure of system operating time or of software execution time.

$\hat{x}$	Estimated value of x
$P\{a\}$	Probability of event a
$P\{a\|b\}$	Probability of event a, conditional on event b
pdf(x)	Probability density function of RV x
cdf(x)	Cumulative distribution function of RV x
$E\{x\}$	Expected value of RV x
$E\{x\|b\}$	Expected value of RV x, conditional upon event b
T, t	Time to next failure
T_i, t_i	Inter-failure time between failures $(i-1)$ and i
X, x	Accumulated operating time
U_j, u_j	In the context of failure count data, operating time in period j
u_i	In the context of predictive accuracy assessment, statistic $u_i = \hat{F}_i(t_i)$, where $\hat{F}_i(t)$ is predictor CDF of the ith TTF
N, n	Number of faults in the system
K_j, k_j	Number of failures in period j
C, c	Cumulative number of failures
M, m	Expected cumulative number of failures, $E\{C\}$
Λ, λ	Rate of occurrence of failure
Z, z	Activation rate of individual fault
S	Survival probability (i.e. reliability)
$a!$	Factorial a (not necessarily an integer): $\int_0^\infty y^a \exp(-y)\mathrm{d}y$ Gamma notation is often used: $\Gamma(a) = (a-1)!$

gamd (z, α, β)	Gamma density function for RV Z: $\beta^{\alpha} z^{(\alpha-1)} \exp(-\beta z)/(\alpha - 1)!$ α, β are the shape and scale parameters, respectively.
$U(0,1)$	Uniform distribution on interval (0,1)
$N(\mu, \sigma)$	Normal distribution with mean μ, and standard deviation σ
$N(0,1)$	Standard normal distribution
$\binom{n}{c}$	Binomial coefficient: number of possible selections of c items from n.
$\underline{p}$	Vector of parameters

B.3 General statistical techniques

B.3.1 *Graphical analysis*

B.3.1.1 *Time to failure data*

This type of data consists of a sequence of c inter-failure times $\{t_1, ..., t_c\}$, possibly followed by a failure-free interval of length t_e up to the end of the observation. t_i denotes the inter-failure time, and x_i the accumulated time, up to failure number i. $c(x)$ denotes the count of failures at accumulated operating time x. c denotes the total count of failures, and x_e the total operating time, up to the end of observation.

The data set may be partitioned into successive periods j of equal amounts of operating time u. k_j denotes the count of failures in period j, x_j the accumulated time, and $c(x_j)$ the accumulated failure count up to the end of period j. $k_j = c(x_j) - c(x_{j-1})$. (If $k_j = 0$, then successive periods may be combined until a non-zero count is obtained, at the expense of having periods of unequal lengths u_j.) The data set may also be partitioned into p periods j containing equal failure counts k, so that

$$[0,x_e] = \bigcup_{j=1}^{p-1} [x_{k(j-1)}, x_{kj}] \bigcup [x_{k(p-1)}, x_e]$$

where x_{kj} denotes the time of kjth failure. (Strictly speaking, each interval apart from the first is open on the left.) $x_0 = 0$, and the last interval contains $[c - k(p - 1)]$ failures, which may be less than k.

The following graphs may be useful:

a) cumulative failure counts:
   1) $c(x_i)$ against x_i;
   2) log $[c(x_i)]$ against x_i;

b) empirical failure rate (ROCOF):
   1) k_j/u against x_j;
   2) log (k_j/u) against log (x_j);
   3) $k/(x_{kj} - x_{k(j-1)})$ against x_{kj} under the partition into equal failure counts k described above. $x_0 = 0$, and the last point is $\{[c - k\ (p - 1)/[x_e - x_{k(p-1)}], x_e\}$;
   4) logarithmic version of **B.3.1.1**b)3);

c) empirical cumulative MTTF:
   1) i/x_i against i;
   2) i/x_i against x;
   3) log (i/x_i) against log (x_i);

d) empirical instantaneous MTTF:
   1) $1/t_i$ against i;
   2) $1/t_i$ against x_i;
   3) log $(1/t_i)$ against log (x_i);
   4) reciprocal of empirical failure rate in **B.3.1.1**b)3) above against x_{kj};
   5) logarithmic version of **B.3.1.1**d)4).

B.3.1.2 *Failure count data*

Here, the observations consist of failure counts k_j, and amounts of use u_j, in each of several successive periods j. c_j is the accumulated count of failures and x_j the accumulated time, up to the end of period j. (If k_j or $u_j = 0$ for any period, successive periods should be combined and renumbered, so that all failure counts and times are non-zero.) The following graphs may be useful.

a) cumulative failure counts:
   1) c_j against x_j;
   2) log (c_j) against log (x_j);

b) empirical failure rate (ROCOF):
   1) k_j/u_j against x_j;
   2) log (k_j/u_j) against log (x_j);

c) empirical MTTF in each period:
   1) k_j/u_j against x_j;
   2) log (k_j/u_j) against log (x_j).

B.3.1.3 *Isotonic regression*

Isotonic regression is performed on the graph of accumulated failure count plotted against accumulated operating time. It consists of drawing a set of straight line segments to envelope the data plot, in such a way that each line touches the plot at two points and lies completely above it. The segments constitute the *least concave majorant* (LCM) for the data set and the slope of the lowest segment above any point is the 'best' estimate of the failure rate at that point. The result is a sequence of estimated rates $\hat{\lambda}_s$ for the successive intervals s topped by each of the segments, subject to the monotonicity constraint that $\hat{\lambda}_s \geq \hat{\lambda}_{s+1}$ for all values of s. This is illustrated in figure 6.

If the data set is terminated by a period of failure-free operation, one failure should be added at the end to avoid a segment with zero slope. This graphical procedure is equivalent to monotonic regression (see **B.3.3** below) with a maximum difference degree of 1.

B.3.2 *Exploratory data analysis (EDA)*

B.3.2.1 *Techniques used in EDA*

EDA is not a single method. It uses general-purpose statistical techniques to search for features of the data, making few assumptions about the underlying process.

Some of the techniques test hypotheses. The general procedure is as follows.

a) Define the null hypothesis H_0 and its alternative(s).

b) Define the level of significance α at which H_0 is to be rejected.

c) Define the test statistic T, and the rejection region R for the distribution of T given H_0.

d) Evaluate T from the data and reject H_0 if T is in R.

Examples of features of the data are trend, independence and randomness. Each has its associated null hypothesis and statistics.

Some of the tests described here are also of use in assessing predictive accuracy of a model (see annex C).

B.3.2.2 *Trend analysis*

The *Laplace test* is applied to a sequence of inter-failure times $\{t_1, \ldots, t_c\}$. x_e denotes the total time of observation. Note that, if this ends with a period of failure-free operation, then

$$x_e > x_c = \sum_{i=1}^{c} t_i$$

The procedure is as follows.

a) H_0 is 'no trend' (sequence $\{t_i\}$ is HPP). H_{RG} is 'reliability growth' (sequence $\{t_i\}$ is stochastically increasing). H_{RD} is 'reliability decay' (sequence $\{t_i\}$ is stochastically decreasing).

b) Statistic $L = \left[\sum_{i=1}^{c} x_i - x_e c/2\right] / \left[x_e\sqrt{c/12}\right]$

c) Under H_0: $L \rightarrow N(0,1)$ as $c \rightarrow \infty$

d) Where $K_{1-\alpha/2}$ is the $(1-\alpha/2)$ percentile of the normal distribution, this leads to conclusions as follows.

$|L| < K_{1-\alpha/2} \Rightarrow H_0$.

$L \leq -K_{1-\alpha/2} \Rightarrow H_{RG}$.

$L + K_{1-\alpha/2} \Rightarrow H_{RD}$.

The Laplace test is adequate at a 5 % level of significance for $c \geq 4$.

Another test is the *MIL-HDBK 189 test*, for which the procedure is as follows.

1) H_0, H_{RG}, and H_{RD} are as for Laplace.

2) Statistic $MH189 = 2\sum_{i=1}^{c} \ln(x_e/x_i)$

3) Under H_0, MH189 is distributed as χ^2 with $2(m-1)$ degrees of freedom.

4) Reference to values of χ^2 is used to accept or reject H_0 in favour of H_{RG} or H_{RD} at the desired level of significance.

The MIL-HDBK 189 test is optimum against trend as exhibited under the Duane type of NHPP model (see below).

A further technique is to attempt to fit an NHPP model such as the *Duane* (or *power-law*) model. This may be done graphically or computationally as follows.

It has been observed that accumulated failures $c(x)$ or empirical failure rates k_j/u_j tend to lie on a straight line when plotted on log-log paper against accumulated operating time x. This gives the relationship:

$$m(x) = \alpha x^{\beta}$$

where $m(x)$ is the expected accumulated number of failures observed at time x. log (α) is the intercept, and log (β) the slope, of the regression line of log $[c(x)]$ on log (x). $\beta < 1$ implies growing, $\beta = 1$ implies constant, and $\beta > 1$ implies decaying, reliability.

This technique may be used with 'time to failure' or 'failure count' data.

The regression is sensitive to early data, e.g. a short period of increasing failure rate at the start of observation may result in an estimate of $\beta > 1$.

For 'time to failure' data (where c is the total number of failures, x_i is time up to ith failure, and x_e is time to end of observation, as above) then MLEs of the parameters, where the data are 'time truncated' so that $x_e > x_c$, are as follows.

$$\hat{\beta} = c/\sum_{i=1}^{c} \ln[x_e/x_i],\ \hat{\alpha} = c/x_e^{\hat{\beta}}$$

If the data are 'failure truncated' so that $x_e = x_c$, the estimators are as follows.

$$\hat{\beta} = c/\sum_{i=1}^{c} \ln[x_c/x_i],\ \hat{\alpha} = c/x_c^{\hat{\beta}}$$

Pearson's chi-squared statistic (see **B.3.2.5** below) may be used to test for uniform distribution (i.e. lack of trend) in data, as well as for 'goodness of fit' of more general models.

There are many other tests for trend, and many of the graphical techniques described in **B.3.1** above provide a qualitative assessment of trend.

B.3.2.3 *Independence tests*

Many black box models assume that successive inter-failure times are independent, or that each fault is activated independently of all others. It is therefore important to check that these assumptions are not violated before applying such models.

For 'time to failure' data, a serial correlation test may be applied. A simple graphical technique is to plot points $(t_i, t_{i-\kappa})$, where κ is the lag. Any correlation between TTFs at that lag will be visible as a departure from an even scatter. The serial correlation coefficient ρ_κ for lag κ may also be calculated as follows. (c is the number of TTFs.)

$$\hat{\rho}_\kappa = \frac{\sum_{i=1}^{c-\kappa} t_i t_{i+\kappa} - (c-\kappa)\,\bar{t}_0\bar{t}_\kappa}{\sqrt{\left[\sum_{i=1}^{c+\kappa} t_i^2 - c - \kappa)\,\bar{t}_0^2\right]\left[\sum_{i=1}^{c-\kappa} t_{i+\kappa}^2 - (c-\kappa)\,\bar{t}_\kappa^2\right]}}$$

where

$$\bar{t}_0 = \sum_{i=1}^{c-\kappa} t_i / (c-\kappa),\ \bar{t}_\kappa = \sum_{i=1}^{c-\kappa} t_{i+\kappa} / (c-\kappa)$$

$-1 \le \hat{\rho}_\kappa \le 1$. When $\hat{\rho}_\kappa = 0$, this implies independence, when $\hat{\rho}_\kappa = 1$, this implies perfect positive correlation and when $\hat{\rho}_\kappa = -1$, this implies perfect negative correlation. Independence is rejected at significance level α if $|\hat{\rho}_\kappa\sqrt{c-1}| > K_{\alpha/2}$, the upper $\alpha/2$ point of the standard normal distribution.

If a data set exhibits reliability growth, this will show as a correlation. Non-stationary behaviour may be removed prior to examining serial correlation. One method is to use the first differences of logarithms of TTF instead of raw TTF, i.e. to plot points $\{[\ln(t_i) - \ln(t_{i-1})], [\ln(t_{i-\kappa}) - \ln(t_{i-\kappa-1})]\}$, or to use the same transformed values to compute $\hat{\rho}_\kappa$.

A further graphical method is to draw a *correlogram*, which is a plot of points $(\kappa, \hat{\rho}_\kappa)$ for $\kappa = 1, \ldots c/4$ (i.e. up to the greatest integer $\le c/4$). If the correlogram is based on raw TTFs exhibiting reliability growth then $\hat{\rho}_\kappa$ will be large for small κ, and decrease linearly as κ increases. A correlogram based on TTFs from which non-stationary behaviour has been removed and in which no serial correlation is present will show random scattering about zero, with all $\hat{\rho}_\kappa$ lying within the approximate confidence interval $\pm 2/\sqrt{c}$.

In some cases, it may be meaningful to adapt these procedures to apply to the empirical TTF or failure rate for 'failure count' data.

B.3.2.4 *Tests for randomness*

Tests for randomness are applied either when a set of raw data is expected to consist of randomly distributed values, or when a transformation which should remove non-random variation from the data has been applied, so that the residuals should be random.

Reliability growth data usually exhibits trend, and so would not be expected to be random. However, it may be transformed to remove trend, e.g. by the log-difference method described for serial correlation tests in **B.3.2.3**. Also, the predictions from reliability growth models may be used to generate statistics that should be randomly distributed if the predictions are accurate. Important examples of the last category are the u and y statistics used to check the accuracy of the predictor CDF $\hat{F}_i\ (t)$ derived from preceding TTFs $\{t_1, \ldots, t_{i-1}\}$.

NOTE. u in this context should not be confused with u as a measure of operating time in a given period.

Realizations t_i of the RVs T_i are substituted into the formulae for CDFs $\hat{F}_i\ (t)$ for i greater than a certain starting value s to obtain statistics as follows.

a) $u_i = \hat{F}_i\ (t_i)$ for $s < i \le c$

b) $x_i = -\ln(1 - u_i)$ for $s < i \le c$

c) $y_i = \sum_{j=s+1}^{i} x_j \,/\, \sum_{j=s+1}^{c} x_j$ for $s < i < c$

If the $\hat{F}_i\ (t)$ are accurate, then the u_i and y_i are realizations of U(0,1) IID RVs, and the x_i constitute a realization of an HPP (see annex C for further details).

The following tests for randomness are described for an arbitrary set of quantities $\{q_1, \ldots, q_m\}$. In each case the assumption of randomness is the null hypotheses H_0.

Assume that the values of q_i are normalized in the range [0,1] and ranked $q_1 \le q_2 \le \ldots \le q_m$. The *Kolmogorov distance* provides an assessment of the closeness of the distribution to U(0,1) as follows.

1) The empirical sample CDF is:

$$F_m(q) = \begin{cases} 0, & q<q_1 \\ j/m, & q_j \le q < q_{j+1} \\ 1, & q > q_m \end{cases}$$

2) The Kolmogorov distance is:

$D_m = \max_i \{|F_m(q_i) - q_{i-1}|, |F_m(q_i) - q_i|\}$

3) If $D_m > D_m{}^\alpha$, then randomness is rejected at significance level α where $D_m{}^\alpha$ is the $(1-\alpha)$ percentile of the Kolmogorov distribution.

A method of assessing this distance graphically using a probability plot or 'u-plot' is described in annex C.

The *Cramer-von-Mieses distance* also assesses the closeness of the distribution to $U(0,1)$ as follows.

i) Distance $M_m = \frac{1}{2m} + \sum_{i=1}^{m}\left[\frac{2i-1}{2m} - q_i\right]^2$

ii) If $M_m > M_m{}^\alpha$, then randomness is rejected at significance level α, where $M_m{}^\alpha$ is the $(1-\alpha)$ percentile of the Cramer-von-Mieses distribution.

Many other statistical tests of randomness are available.

B.3.2.5 *Tests for goodness of fit*

Tests of this type assess the level of significance at which observations agree with the predictions of a model. The χ^2 test is frequently used for this purpose.

Pearson's chi-squared statistic provides an approximate significance test where there are k expected values E_i (predicted by the model) and corresponding observed (discrete) values O_i.

The test assumes that there are $n = \sum_{i=1}^{k} O_i$ underlying 'trials', and a complete set of k events $\{A_i\}$ with a set of probabilities $\{\theta_i\}$ of 'success of trial' associated with each, so that each O_i is an observed realization of a RV N_i, the 'count of successful trials', where:

$E\{N_i\} = n\theta_i$, and $\sum_{i=1}^{k} \theta_i = 1$.

a) H_0 is 'the model adequately describes the observations'. (The model assigns a set of values $\{p_i\}$ to the set of probabilities $\{\theta_i\}$, so that $E_i = E\{N_i|H_0\} = np_i$.)

b) Pearson's statistic $\chi^2 = \sum_{i=1}^{k} (O_i - E_i)^2 / E_i$

c) For large n, Pearson's statistic approximately follows a χ^2-distribution with $(k - 1)$ degrees of freedom. (The distribution is strictly followed only as $n \to \infty$. A rule of thumb for the test to be valid is that $E_i \geq 2$ for all i.)

d) H_0 is rejected at significance level if $\chi^2 > \bar{\chi}_a^2$ $(k - 1)$, the upper 100α% point of the χ^2 distribution with $(k - 1)$ degrees of freedom.

Pearson's test may be applied to a continuous RV X whose distribution is predicted by a model by defining A_i as $a_i \leq x < a_{i+1}$, so grouping the observations into a set of intervals such that $p_i = \int_{a_i}^{a_{i+1}} \text{pdf}(x)\text{d}x$. (This can easily be extended to the case where each observation is of the realizations of several RVs.)

An important special case is that of a prediction of a $U(0,1)$ distribution. This may be tested by partitioning [0,1] into k intervals of equal length $1/k$, and using $E_i = c/k$, where c is the total number of observations. This may be used to test the uniformity of the statistics u_i and y_i defined in **B.3.2.4**.

The *Braun statistic* (see annex C) may also be used to assess the accuracy of certain predictions.

B.3.3 *Monotonic regression*

The necessary and sufficient condition for a function $\lambda(x)$ to be completely monotone is that it possesses derivatives of all orders, and $(-1)^n \text{d}^n\lambda(x)/\text{d}x^n \geq 0$, where $x \geq 0$, $n \geq 0$.

Any completely monotone function may be used as the failure intensity function (see **B.4.1.1**) of a doubly-stochastic EOS model (see **B.4.1.3** and **B.4.1.4**). Conversely, any EOS intensity function is completely monotone.

A set of failure data can be analysed using the monotonicity of the hypothesized underlying failure intensity. The technique can be applied to 'failure count' data, or to 'time to failure' data which has been divided into p periods of equal time u (see **B.3.1.1**).

A completely monotone function is fitted to the sequence of monotonically decreasing failure rates by defining a set of difference operators D_e of various degrees e up to some maximum d as follows:

$D_0(\lambda_j) = \lambda_j = k_j/u_j = [c(x_j) - c(x_{j-1})]/u_j$

$D_1(\lambda_j) = D_0(\lambda_j) - D_0(\lambda_{j-1}) = \lambda_j - \lambda_{j-1}$

$D_2(\lambda_j) = D_1(\lambda_j) - D_1(\lambda_{j-1})$

$D_d(\lambda_j) = D_{d-1}(\lambda_j) - D(\lambda_{j-1})$

Estimated rates $\hat{\lambda}$ which constitute the 'best fit' are obtained by minimizing the sum of the squared deviations $[\lambda_j - \hat{\lambda}_j]^2$ over all periods j, $1 \leq j \leq p$, subject to the monotonicity constraints:

$(-1)^e D_e(\hat{\lambda}_j) \geq 0$, for all e, $1 \leq e \leq d$

For $d = 1$, the technique is equivalent to *isotonic regression* (see **B.3.1.3**), and can be performed graphically. In this case, the rate at the end of the data set tends to be underestimated. A degree d of 2 or 3 has been found to give less biased estimates. For degree $d > 1$, numerical optimization is required, and will generally require a computer program.

B.4 Black-box models

B.4.1 *Black-box reliability assessment*

Any black-box model comprises three parts as follows.

a) A *probabilistic model*: a set of formulae for reliability functions such as $m(x)$, $\lambda(x)$, or $S(t \mid \underline{c},x)$ incorporating a set of parameters $\underline{p}$.

b) An *inference procedure* to estimate the values of $\underline{p}$ by analysis of failure data.

c) A *prediction procedure* to combine the values of $\underline{p}$ with the formulae to predict future times to failure or failure counts.

Together these constitute a *prediction system*, and any part may affect the quality of prediction from the whole.

The data consists of a set of observed inter-failure times $\{t_1, t_2, ..., t_i\}$ or of observed counts of failures $\{k_1, k_2, ..., k_j\}$ in periods of operation in which system use was $\{u_1, u_2, ..., u_j\}$. The task is to obtain CDFs or PDFs for the future times to failure (RV) $\{T_{i+1}, T_{i+2}, ...\}$ or failure counts (RV) $\{K_{j+1}, K_{j+2}, ...\}$, or the means, variances, medians and percentiles of these distributions.

The inference procedure (see **B.4.5**) may employ several methods such as MLE (which searches for values to maximize the probability of the observations), or LSE (which searches for values to minimize the LSD between observed and expected values).

The prediction procedure may involve the simple substitution of the inferred parameter values in the model formulae, but in some cases Bayesian estimation can be used.

B.4.2 *Fault activation models*

B.4.2.1 *EOS models*

Most fault activation models assume that each individual fault has an exponential time to activation, and that the process of failure observed from the whole system consists of the order statistics of the activation processes, hence such models are known as exponential order statistic (EOS) models. It is useful to consider individual models as special cases of a general EOS model.

The assumptions of the general EOS model are:

a) the system contains a set of faults initially;

b) each fault gives rise to failure (i.e. is activated) independently of all others;

c) each fault is activated with its own characteristic rate;

d) the process of activation of each fault is a homogenous Poisson process (HPP), i.e. it is constant over time, and inter-activation times are exponentially distributed RVs;

e) on activation, a fault is immediately and perfectly removed from the system.

The failure process of the system is therefore determined by the set of activation rates $\{z_1, z_2, ...\}$.

Without loss of generality, it can be assumed that:

1) $z_1 \geq z_2 \geq ...$ etc.;

2) there are infinitely many faults. The case of a finite number, n, of faults is dealt with by assuming $z_i = 0$ for $i > n$.

An EOS model is uniquely defined by any one of the following four functions:

i) mean value function, $m(x)$: expected number of faults activated by accumulated time x;

ii) failure intensity function, $\lambda(x)$: expected ROCOF at accumulated time x, given the expected number of faults activated by that time. $\lambda(x) = m'(x)$;

iii) activation rate distribution function, $G(z)$: number of faults with rate $z_i > z$. (Where the number of faults is treated as a RV N, or the activation rates are treated as realizations of a RV Z, $G(z)$ is the expected number.);

iv) activation rate generating function, $g(z)$: $g(z) = -G'(z)$.

These are related by the following equations:

$$G(z) = \int_z^\infty g(z)\mathrm{d}\, z \qquad \text{(B.1)}$$

$$m(x) = \int_0^\infty (1 - \exp(-zx))g(z)\mathrm{d}\, z \qquad \text{(B.2)}$$

$$\lambda(x) = \int_0^\infty z\, \exp(-zx)g(z)\mathrm{d}\, z \qquad \text{(B.3)}$$

The ROCOF $\lambda(x)$ above is not conditional on the number of faults removed, but only on the total time x. Other formulæ can be derived for the hazard intensity after time x and c faults found for particular models.

B.4.2.2 *Deterministic EOS models*

In a deterministic EOS model (DET/EOS) model the set of activation rates $\{z_i\}$ is specified by a deterministic formula, instead of being defined randomly. $G(z)$ is therefore the actual number of faults with rates $z_i > z$.

The best known example of a DET/EOS model is Jelinski-Moranda (JM). It assumes a fixed but unknown number of faults, n, each with the same activation rate φ, so that ROCOF is proportional to the number of remaining faults.

Hence:

$$z_i = \varphi \text{ for } 1 \leq i \leq n; \qquad z_i = 0 \text{ for } i > n$$

$$G(z) = n \text{ for } 0 \leq z \leq \varphi; \qquad G(z) = 0 \text{ for } z > \varphi$$

$$m(x) = n[1 - \exp(-\varphi x)]$$

$$\chi(x) = \varphi\, n\, \exp(-\varphi x) = [n - m(x)]\, \varphi$$

PDF of time to next failure t, after c faults have been activated, is given by:

$$\mathrm{pdf}(t|c) = \varphi\, (n - c)\exp[-\varphi\, (n - c)t]$$

and survival probability (reliability) is as follows:

$$S(t|c) = 1 - \exp\, [-\varphi\, (\mathrm{n} - c)\, t]$$

Hazard intensity is constant between failures.

$$\chi(c) = (n - c)\, \varphi$$

$$\mathrm{MTTF} = E\, \{T|c\} = 1/\lambda(c) = 1/(n - c)\varphi$$

The pth percentile, t_p, of the distribution of TTF is as follows.

$$t_p = \frac{-\ln(1 - p)}{\varphi(n - c)t}$$

LF for 'time to failure' data is as follows.

$$\mathrm{LFT}(n, \varphi\, |t_1, ...,t_c)$$

$$= \prod_{i-1}^{c} \varphi\, (n-i + 1)\exp[-\varphi(n - i + 1)t]$$

B.4.2.3 *IID doubly-stochastic models*

These are classified in **6.4** as 'random activation rate' models. Since they treat individual fault activation rates as realizations of independent identically distributed random variables, they are referred to as IID/EOS models. They make the same set of assumptions as all EOS models, together with the following.

a) The set of faults initially present in the system contains a fixed but unknown number of faults, n.

b) Each activation rate z_i of an individual fault is a realization of a RV Z_i. Each Z_i is independent of all others, with PDF pdf(z) (the same for all Z_i), i.e. the Z_i are IID RVs.

The hazard intensity after c of the initial n faults have been found is therefore a RV Λ.

$$\Lambda = \sum_1^{n-c} Z_i$$

For an IID/EOS model, the fault generating function is $g(z) = n$ pdf(z).

An example of an IID/EOS model is the Littlewood stochastic reliability growth (LSRG) model. In LSRG, the common PDF of the Z_i is gamd(z, α, β), hence:

$$g(z) = n\beta^{\alpha}\, z^{(\alpha-1)}\, \exp(-\beta z)/(\alpha - 1)!$$

The model has three parameters, n, α and β, where α and β are the shape and scale parameters of the gamma distribution.

The nature of the activation rate distribution is determined by α. For $\alpha < 1$, gamd(z, α, β) tends to infinity as $z \to 0$. This models the case of very many faults with very small activation rates. For $\alpha > 1$, the gamma distribution is unimodal, which models the case of rates clustering around some moderate value.

From equations (B.2) and (B.3) the expected failure count and ROCOF are as follows.

$$m(x) = n\left[1 - \left(\frac{\beta}{\beta + x}\right)^{\alpha}\right]$$

$$\lambda(x) = n\left(\frac{\alpha}{\beta}\right)\left(\frac{\beta}{\beta + x}\right)^{\alpha+1}$$

$$= \frac{n\alpha\beta^{\alpha}}{(\beta + x)^{\alpha+1}} = n\left(\frac{\alpha}{\beta + x}\right)\left(\frac{\beta}{\beta + x}\right)^{\alpha}$$

Survival probability (reliability) after time x (not conditioned on count c of faults found) is as follows.

$$S(t|x) = \left[1 - \left(\frac{\beta}{\beta + x}\right)^{\alpha} + \left(\frac{\beta}{\beta + x + t}\right)^{\alpha}\right]$$

Since when IID RVs with gamma distributions are added, the shape parameters are added to obtain the distribution of the sum, it follows that, after c of the initial n faults have been removed, Λ is distributed as follows.

$$\text{pdf}(\lambda) = \text{gamd}[\lambda,(n - c)\alpha,\beta]$$

Another basic result is that the PDF of time to activation for faults remaining after operating time x is as follows.

$$\text{pdf}(z|\text{fault not activated in } [0,x])$$

$$= \frac{(\beta + x)^{\alpha}\, z^{\alpha-1}\, \exp[-(\beta + x)z]}{(\alpha - 1)!}$$

$$= \text{gamd}(z,\ \alpha,\ \beta + x)$$

This represents mathematically the intuitive confidence that, the longer a system has been operated without failure, the smaller the fault activation rates are likely to be. (Note that the average rate of activation of an individual fault is α/β initially, and $\alpha(\beta + x)$ after time x.)

From these results, the following equations can be derived for the failure intensity, survival probability (i.e. reliability), PDF of time to failure, and MTTF, during a future period of operation of length t, conditional on the system already having been used for time x during which c faults have been removed.

$$\lambda(t|x,\ c) = \frac{(n - c)\alpha}{(\beta + x + t)}$$

(The behaviour of this failure intensity function is illustrated in figure 10.)

$$S(t|c,\ x) = \left[\frac{\beta + x}{\beta + x + t}\right]^{(n-c)\alpha}$$

$$\text{pdf}(t|c,\ x) = \frac{(n - c)\alpha\ (\beta + x)^{(n-c)\alpha}}{(\beta + x + t)^{(n-c)\alpha+1}}$$

$$\text{MTTF} = \text{E}\{T|c,\ x\} = \frac{(\beta + x)}{(n - c)\alpha - 1}$$

Time to failure T follows a Pareto distribution and does not always possess moments. If $n - c \leq 1/\alpha$, the equation for MTTF is meaningless. MeTTF always exists.

The LF in the case of 'time to failure' data at the cth failure is as follows.

$$\text{LFT}(n,\alpha,\beta) = \prod_{i=1}^{c} \text{pdf}(T_i|t_1,...,t_{i-1})$$

$$= \frac{\prod_{i=1}^{c} (n - i + 1)\alpha(\beta + x_{i-1})^{(n-i+1)\alpha}}{(\beta + x_{i-1} + t_i)^{(n-c+1)\alpha+1}}$$

For 'failure count' data, the LF is as follows.

P {any given fault *not* activated in interval i}

$= q_i = [(\beta + u_{i-1})/(\beta + u_i)]^{\alpha}$

P {any given fault *is* activated in interval i}

$= p_i = 1 - q_i$

The LF for m intervals is as follows

$$\text{LFC}(n,\ \alpha,\ \beta) = \prod_{i=1}^{m} \left(\frac{n - c_{i-1}}{k_i}\right) p_i^{ki}\ q_i^{(n-c_i)}$$

B.4.2.4 *NHPP doubly-stochastic models*

These models are referred to in **6.4** as 'random number of faults' models, since some of them model the 'number of faults in the system' as a RV N with a Poisson distribution. However, a NHPP model also results if the activation rates z_i are modelled as a realization of a NHPP with intensity function $g(z)$, which is the fault generating function in this case.

In either case, the process of system failure over time x is modelled as a NHPP with intensity $\lambda(x)$.

An example of a NHPP model is Goel-Okumoto (G-O). This was originally derived as a 'failure count' model, based on the assumptions that the cumulative number of faults activated by time t is a RV with a Poisson distribution with mean $m(t)$ and that the number of faults activated in a short interval is proportional to the number of faults present at the start of the interval. With the boundary condition $m(0) = 0$, the mean value function:

$$m(x) = \overline{N}[1 - \exp(-\varphi x)]$$

is obtained as the solution of the differential equation for the failure intensity:

$$\lambda(x) = m'(x) = \varphi[\overline{N} - m(x)]$$

where $\overline{N}$ is the expected value of RV N and $m(x) \to \overline{N}$ as $x \to \infty$. As in J-M, $z_i = \varphi$ for all i.

The likelihood function based on p periods of observation is as follows.

$$\text{LFC}(\overline{N}, \varphi) = \prod_{j=1}^{p} \frac{[m(x_j) - m(x_{j-1})]^{kj} \exp[m(t_{j-1}) = - m(t_j)]}{k_j!}$$

A NHPP model that avoids the 'uniform activation rate' assumption made implicitly by G-O is Musa-Okumoto (M-O), which assumes that hazard intensity decrement per fault removed decreases exponentially with cumulative faults found:

$$\lambda(c) = \lambda_0 \exp[-\theta c] = \lambda_0 [\exp(\theta)]^{-c}$$

where λ_0, the initial hazard intensity, and θ are the parameters of the model. The mean value function (expected failure count at time x) is as follows.

$$E\{C(x)\} = m(x) = \frac{1\text{n}[\lambda_0 \theta x + 1]}{\theta}$$

The count of failures $c(x)$ at time x follows a Poisson distribution.

$$P\{c(x)\} = \frac{[m(x)]^{c(x)} \exp(- m(x))}{c(x)!}$$

The unconditional ROCOF is as follows.

$$\lambda(x) = m'(x) = \frac{\lambda_0}{\lambda_0 \theta x + 1}$$

(The same formula for $\lambda(x)$ is obtained if the formula for $m(x)$ is substituted for c in the expression for $\lambda(c)$ above, i.e. $\lambda(x)$ is the expected ROCOF.)

Conditional on the previous failure having occurred at time x_{i-1}, the reliability (survival probability), hazard intensity, and MTTF describing the behaviour over TTF T_i are as follows.

$$S(t_i|x_{i-1}) = \left[\frac{\lambda_0 \theta x_{i-1} + 1}{\lambda_0 \theta (t_i + x_{i-1}) + 1}\right]^{1/\theta}$$

$$\lambda(t_i|x_{i-1}) = \frac{\lambda_0}{\lambda_0 \theta (t_i + x_{i-1}) + 1}$$

$$E\{T_i|x_{i-1}\} = \frac{\lambda_0 \theta x_{i-1} + 1}{\lambda_0 (1 - \theta)}$$

For 'time to failure' data, the LF conditional on c failures observed at TTFs $\{x_i\}$ and total time at end of observation x_e is as follows.

$$\text{LFT}(\theta, \lambda_0 | x_1, ..., x_c, x_e) = \frac{c!}{[m(x_e)]^c} = \prod_{i=1}^{c} \lambda(x_i)$$

$$= \frac{c!(\lambda_0 \theta)^c}{[1\text{n}(\lambda_0 \theta x_e + 1)]^c \prod_{i=1}^{c} (\lambda_0 \theta x_i + 1)}$$

For 'failure count' data, the LF conditional on c failures observed in p periods with failure count k_j in each period $(x_{j-1}, x_j]$ is as follows.

$$\text{LLC}(\theta, \lambda_0 \mid c, \{k_j, x_j\})$$

$$= \frac{c!}{[m(x_p)]^c} \prod_{j=1}^{p} \frac{[m(x_j) = m(x_{j-1})]^{k_j}}{k_j}$$

$$= \frac{c!}{[1\text{n}(\lambda_0 \theta x_p + 1)]^c} \prod_{j=1}^{p} \frac{1}{k_j} \left[1\text{n}\left(\frac{\lambda_0 \theta x_j + 1}{\lambda_0 \theta x_{j-1} + 1}\right)\right]^{k_j}$$

The above LFs may be transformed by substituting $\varphi = \theta \lambda_0$ and taking logarithms. By setting the first derivatives of the LLFs so obtained equal to zero, equations are obtained which can be solved numerically to give a point estimate $\hat{\varphi}$, and $\hat{\theta}$ (and hence $\hat{\lambda}_0$) is derived from the mean value function.

$$\hat{\theta} = 1\text{n}\ (\hat{\varphi} x_e + 1) / c \text{ or } 1\text{n}\ (\hat{\varphi} x_p + 1) / c$$

$$\hat{\lambda}_0 = \hat{\varphi} / \hat{\theta}$$

Generally, a NHPP/EOS model may be constructed by hypothesizing any suitable intensity function (or mean value function) for the NHPP. In particular, there is a NHPP/EOS model corresponding to each DET/EOS or IID/EOS model, e.g. G-O is a NHPP version of J-M.

In the Duane (DU) model, the functions are:

$$m(x) = \alpha x^{\beta}$$

$$\lambda(x) = m'(x) = x\beta x^{\beta - 1}$$

where α and β are parameters. (Methods of estimating these are discussed in **B.3.2.2**.)

The Poisson-gamma (PG) model is the NHPP version of LSRG, with intensity function

$$\lambda(x) = n\alpha\beta^{\alpha}/(\beta + x)^{\alpha+1}$$

B.4.3 *Failure trend models*

Models in this class represent the inter-failure times directly as a parametric family of functions. An example is the Littlewood-Verrall (L-V) model. This assumes that the time to failure T_i after each failure i is an exponential RV with intensity Λ_i, which is itself an RV with a gamma distribution.

$$\text{pdf}\ (t_i|\Lambda_i = \lambda_i) = \lambda_i \exp(-\lambda_i t_i)$$

$$\text{pdf}(\lambda_i) = \text{gamd}\ (\lambda_i, \alpha, \beta(i))$$

The T_i are assumed to be independent. The scale parameter $\beta(i)$ of the gamma distribution determines reliability growth (if increasing in i) or decay (if decreasing in i). A parametric family $\beta(i) = \beta_1 + \beta_2 i$ is hypothesized so that growth or decay depends on the sign of β_2. (α, β_1, β_2 are the three parameters of the model.)

Other parametric families for $\beta(i)$ are possible. The model assumes two sources of uncertainty: failures occur randomly, and a repair action following a failure only causes a decrease in failure rate with a certain probability, so allowing the possibility of imperfect repair. The criterion for reliability growth *on average* is:

$$P\{\lambda_{i+1} < \rho\} \geq P\{\lambda_i < \rho\} \text{ for all } i, \rho > 0$$

Combining these two types of uncertainty using Bayesian techniques, a distribution of T_i is obtained which is Pareto in form.

$$\text{pdf}(t_i) = \frac{\alpha[\beta_1 + \beta_2 i]^{\alpha}}{[t_i + \beta_1 + \beta_2 i]^{\alpha+1}}$$

Hazard intensity after i failures is decreasing in t.

$$\lambda(t|i) = \alpha/(t + \beta(i))$$

Survival probability (reliability) has the following form.

$$S(t|i) = [\beta(i)/(t + \beta(i))]^{\alpha}$$

$$\text{MTTF} = E\{T_{i+1}|c\} = \beta(i)/(\alpha - 1)$$

This exists if, and only if, $\alpha > 1$.

The pth percentile t_p of the distribution of TTF is as follows:

$$t_p = \beta(i)/[(1-p)^{-1/\alpha} - 1]$$

The likelihood function (for 'time to failure' data, with a total of c failures) is:

$$\text{LF}(\alpha, \beta_1, \beta_2) = \frac{\alpha^c \prod_{i=1}^{c} [\beta_1 + \beta_2 i]^{\alpha}}{\prod_{i=1}^{c} [t_i + \beta_2 + \beta_2 i]^{\alpha+1}}$$

B.4.4 *Environmental factors models*

Accounting for environmental factors requires a model whose estimates can be modified using measurements of those factors. The proportional intensity approach hypothesizes that the system possesses a 'baseline' failure intensity λ_0 which is modified by an expression incorporating such measurements to give the actual rate λ:

$$\lambda(x, c, \underline{p}) = \lambda_0(x, c, \underline{p}) \exp\left(\sum_{i=1}^{v} \beta_i e_i\right)$$

where:

a) each of the v 'explanatory variables' e_i is a measurement of the deviation of a defined attribute of the operating environment from some 'nominal' value. These may be real numbers, integers, or (to indicate that a factor is either present or absent) binary (0 or 1), and are evaluated by observation of the environment independently of the observation of the failure of the system;

b) each quantity b_i is a parameter of the model and represents the extent to which the corresponding e_i affects the failure rate. (It is a weighting factor.) The b_i are estimated from observation of system failure in environments for which the e_i are known. (A zero value obtained for a b_i indicates that the factor measured by e_i has no effect.);

c) $\lambda_0(x, c, \underline{p})$ is the 'baseline' failure rate which would be obtained if all factors were nominal, i.e. if all e_i were zero. This is also measured by observation of system failure in known environments. (It may vary over operating time x, with faults found c, and/or with respect to a set of other parameters $\underline{p}$, such as occur in stochastic reliability growth models.);

d) $\lambda(x, c, \underline{p})$ is the failure rate expected to be observed in an environment characterized by the given set of e_i (and may also vary with x, c, and $\underline{p}$).

The baseline hazard function may be distribution-free, or may follow a distribution defined by parameters. In particular, a version of PHM has been proposed in which the baseline function is the hazard function of one of the stochastic reliability growth models.

B.4.5 *Black-box parameter inference*

B.4.5.1 *Types of inference method*

The second part of a black-box prediction system (see **B.4.1**) is the inference procedure which calibrates the model to a data set by evaluating the parameters. The methods commonly used are described in general terms in **6.4.10.2**. They are as follows.

a) maximum likelihood estimation (MLE) searches for parameter values which maximize the probability of the observations;

b) least squares estimation (LSE) searches for parameter values which minimize the least squared distance (LSD) between the data points and corresponding predicted quantities;

c) Bayesian inference transforms a prior distribution of parameter values into a more accurate posterior distribution using the information contained in the data set;

d) model specific methods take advantage of the mathematical properties of a particular model to derive estimates.

B.4.5.2 *Objective function*

MLE and LSE usually involve optimizing an 'objective function'.

In the case of MLE, this is a likelihood function (LFT in the case of 'time to failure' data, LFC for 'failure count' data). The general form of these is as follows.

$$\text{LFT}(\underline{p}|\{t_i\}_c) = \prod_{i=1}^{c} \text{pdf}(t_i|\underline{p}, \{t_j\}_{i-1})$$

$$\text{LFC}(\underline{p}|\{k_j, u_j\}_p) = \prod_{j=1}^{p} P\{k_i|\underline{p}, u_j, \{k_i, u_i\}_{j-1}\}$$

where $\{t_j\}_c$ denotes the sequence of observations $t_1, \ldots, t_c$ (and similarly for the sequence of pairs of observations $\{k_j, u_j\}_p$) and $\underline{p}$ is the vector of parameters of the model.

Several examples of these are given above. Both a LFT and a LFC can be derived for most models. For optimization, the logarithms of these functions, LLFC or LLFT, are normally used, and the extended products become extended sums.

For LSE, the general form of the distance function is, for time to failure data:

$$\sum_{i=1}^{c} [m(x_i|\underline{p}) - i]^2$$

or, for failure count data:

$$\sum_{i=1}^{p} [m(x_i|\underline{p}) - c(x_i)]^2$$

The empirically estimated $\{\lambda_i\}$ and the expected $\{\lambda(x_i|\underline{p})\}$ may be used similarly.

B.4.5.3 *Search procedure*

In most cases, the LF or DF are too complicated to permit a closed form expression to be derived for the optimum value of $\underline{p}$. (There are exceptions with some simple models, such as Duane. See **B.3.2.2**.)

A numerical search is required over possible parameter values. To reduce the number of dimensions in which the search is performed, it is usual to partially differentiate the objective function with respect to one of the parameters, set the result to zero, and solve the resulting equation for one of the parameters in terms of the others.

The Newton-Raphson algorithm is a procedure which converges rapidly to a minimum value (once in the neighbourhood of such an optimum value) of an objective function f(φ) in a series of iterative steps:

$$\varphi_{n+1} = \varphi_n - f'(\varphi_n) / f''(\varphi_n)$$

This depends upon being able to find a reasonable starting value φ_o such that the iteration does converge. Such a value may be estimated from the data, taking account of model characteristics, and no general rule can be given here.

B.4.5.4 *Bayesian inference*

A prior pdf($\underline{p}$) is transformed into a posterior PDF using Bayes' theorem:

$$\text{pdf}(\underline{p}|\text{data}) = A \text{ pdf } (\underline{p}) \ P\{\text{data}|\underline{p}\}$$

where A is a constant of proportionality:

$$A = 1/\int_{\underline{p}} \text{pdf}(\underline{p}) \ P\{\text{data}|\underline{p}\}d\underline{p}$$

and the integration is over all values of all parameters.

B.4.5.5 *Model specific methods*

An example is a possible method of estimation in M-O (see **B.4.2.4**). This depends upon the form of the basic equation for the hazard intensity after c failures:

$$\lambda(c) = \lambda_o \exp[-\theta_c]$$

This implies that the empirical failure rate $\lambda(i)$ has a log-linear relationship to i:

$$\ln [\lambda(i)] = \ln (\lambda_o) - \theta_i \text{ for all } i$$

The empirical failure rates can be estimated from the data, e.g. as $1/t_i$, and linear regression used to estimate the parameters.

B.4.5.6 *Interval estimates*

The discussion of parameter estimation here has concentrated on making point estimates. In addition, interval estimates are required. However, a presentation of the methods for doing this is beyond the scope of this annex.

B.5 Structural models

B.5.1 *Modular structural models*

Modular structural models treat the execution of software as a process of passing control between a number of discrete parts or 'modules' with certain probabilities of failure occurring on transfer of control or during execution of a module. This is modelled as a markov or semi-markov process.

An example is the Littlewood model. This assumes:

a) the system consists of a finite number of modules, n. At any time, one and only one module is being executed. Probability p_{ij} of transfer of control from module i to module j is independent of the time of entering i (semi-markov property);

b) each module i fails exponentially while execution is within it, at its own rate λ_i;

c) distribution of sojourn time in i before transfer to j depends only on i and j, and its mean μ_{ij} and second moment are known;

d) transfer of control from i to j has a probability v_{ij} of failure.

Assuming that $\lambda_i << \mu_{ij}$ and μ_{ij} is small compared to system operating time for all i, j, then asymptotically as system time increases, the total system failure rate tends to a value λ_S given by:

$$\lambda_S = \sum_{i=1}^{n} a_i \lambda_i + \sum_{i=1}^{n} \sum_{i=1}^{n} b_{ij} v_{ij}$$

where a_i is the limiting proportion of total system operating time spent executing module i, and b_{ij} is the limiting frequency of transfer of control from i to j. a_i and b_{ij} are complicated functions of u_{ij} and p_{ij}.

B.5.2 Hardware/software system models

Many structural software models assume that at any time during system operation one, and only one, module is being executed. In a more general system (e.g. one containing several processors) several modules may be executed simultaneously, and several hardware components will also be active. (It is often assumed that all hardware components are active at all times during operation.) A more general formulation of the problem is therefore necessary for a system including both hardware and software.

One approach is to model the behaviour of the system as a markov chain with S states and mean sojourn time μ_j in state j, $j = 1, ..., S$. Define $p_{jk} = P\{$transition from state j to state $k\}$. (A state transition will involve the start or end of execution of one or more components.) If there are C components and λ_i is the failure rate of component i, and if δ_{ij} takes the value 1 if component i is active in state j, and value 0 otherwise, then the system failure rate in state j, ξ_j is as follows:

$$\xi_j = \sum_{i=1}^{C} \delta_{ij} \lambda_i$$

If $P_j(t) = P\{$system is in state j at time $t\}$, then system failure rate $\lambda(t)$ at time t is as follows.

$$\lambda(t) = \sum_{j=1}^{S} \xi_j P_j(t)$$

Assuming that, in each state, the mean sojourn time is large compared to the time to failure, i.e. $\mu_j >> 1/\xi_j$, and denoting the steady-state probability of being in state j by a_j, then the steady-state failure rate λ of the system is as follows.

$$\lambda = \sum_{j=1}^{S} a_j \xi_j = \sum_{j=1}^{S} a_j \sum_{i=1}^{C} \delta_{ij} \lambda_i = \sum_{i=1}^{C} \lambda_i \sum_{j=1}^{S} \delta_{ij} a_j$$

The term

$$\pi_i = \sum_{j=1}^{S} \delta_{ij} a_j$$

is the average proportion of system execution time for which component i is active, and $\pi_i \lambda_i$ is the contribution of component i to the failure rate of the whole system under the *execution profile* defined by the vector $\Pi = [\pi_i]$. Since the execution profile depends on the *operational profile* of the system, this approach provides another possible starting point for modelling the effect of environment on system reliability.

This formulation is indifferent as to whether any component consists of hardware or software. It is capable of being extended to allow variation of component failure rates over time (e.g. reliability growth of software components) to incorporate hierarchical structure among the components and to allow inclusion of times to restore service (e.g. MTTRS) in order to model system availability. These extensions are beyond the scope of this annex.

Annex C (informative)
Predictive accuracy of stochastic reliability growth models

C.1 Introduction

The software reliability growth models described in **6.4.9** to **6.4.15** are used to predict various measures related to reliability, e.g. probability of system failure within a given time, failure rate, mean time to failure, etc. The accuracy of these predictions should be assessed. If they are found to be biased, it is possible to improve their accuracy by 'recalibrating' them, or by combining predictions from different models.

This annex contains mathematical descriptions of some of the techniques for assessment, recalibration and combination. It is intended to assist users of the standard to apply these techniques (using custom-written or 'off-the-shelf' statistical software as necessary). The descriptions are brief and technical, and reference should be made to **6.4.20** for more general information. Mathematical results are generally quoted without proof, but these are available from the references in the bibliography cited in **6.4.20**.

See annex B for mathematical terms and definitions, and for the mathematical descriptions of the prediction systems. Each prediction system consists of a stochastic reliability model, an inference procedure, and a prediction procedure (see **B.4.1**). Since any one of the three parts may affect the accuracy of the predictions from the whole, the term 'prediction system' is used in this annex in preference to 'model'.

C.2 Assessment of predictive accuracy

C.2.1 *Definition of 'u' and 'y' statistics*

Future time to failure (TTF) is uncertain and is modelled as a random variable (RV) T_i. This has a certain cumulative distribution function (CDF) $F_i(t)$, which is unknown but can be estimated by applying a prediction system to the sequence of previously observed inter-failure times $\{t_1, t_2, ..., t_{i-1}\}$ (assuming 'time to failure' data is available). The result is a predictor CDF $\hat{F}_i(t)$ for the RV T_i , from which other measures such as failure rate, MTTF, MeTTF, etc., can be derived. The criterion for a prediction to be accurate is that the predictor CDF $\hat{F}_i(t)$ should be close to the 'true' CDF $F_i(t)$.

The accuracy of the prediction may be checked by observing a realization t_i of the RV T_i and substituting this into the formula for the predictor CDF $\hat{F}_i(t)$ to obtain the statistic $u_i = \hat{F}_i(t_i)$.

By definition, the true CDF of T_i is $F_i(t) = P\{T_i < t\}$, i.e. the probability that the realized value of T_i will be less than any given value. Hence, if $\hat{F}_i(t)$ is identical to the 'true' CDF $F_i(t)$, of T_i, then u_i is a realization of a RV with a U(0,1) distribution.

In 'one step ahead' prediction of software reliability, only one realization t_i of each RV T_i is observed. (Under reliability growth, $\hat{F}_i(t)$ is different for each i.) However, stochastic reliability models assume that successive TTFs are independent, or that each fault is activated independently of all others. Given this, if the predictor CDF is close to the true CDF for all i, the sequence $\{u_i\}$, $i = s + 1, ..., c$ should constitute a sequence of realizations of IID U(0,1) RVs, and so should be evenly distributed in the interval (0,1) and show no evidence of trend with increasing i. s s)

NOTE. A certain minimum number of data points s (generally around 20) is necessary before the prediction system can be used to begin the 'one step ahead' prediction process, so that if there are c observed t_i, only $(c - s)$ u_i values can be obtained.

To test the sequence of u_i for trend, it may be transformed logarithmically into a sequence of x_i.

$$x_i = -\ln(1 - u_i) \text{ for } s < i \leq c$$

If the u_i are realizations of U(0,1) IID RVs, then the x_i are realizations of exponential IID RVs, i.e. the sequence constitutes a realization of a HPP.

Another transformation may be used to normalize the x_i into a sequence of y_i.

$$y_i = \sum_{j=s+1}^{i} x_j \Big/ \sum_{j=s+1}^{c} x_j \text{ for } s < i < c$$

If the u_i are realizations of U(0,1) IID RVs, then the y_i are the order statistics of $(c - s)$ U(0,1) IID RVs. (Note that only $(c - s - 1)$ values y_i can be obtained.)

Various statistical tests may be used to check that the statistics u_i, x_i and y_i conform to the distributions that are expected if the predictions are accurate.

C.2.2 *Tests for even distribution of u*

Any departure of the u_i from U(0,1) indicates inaccurate prediction. In particular, a preponderance of small values indicates optimistic bias, i.e. the $\hat{F}_i(t)$ tend to predict a TTF larger than the observed t_i. (Conversely, large values indicate pessimistic bias.)

Techniques that may be used to assess this include scatter plots, box plots, u-plots, and the use of Pearson's chi-squared statistic.

A scatter plot is a plot of points (i,u_i). Distribution can be checked simply, e.g. around 50 % of points should lie above $u = 0.5$, around 25 % above $u = 0.75$, etc. Since the points are ordered by time along the horizontal axis, this check may be performed for different time intervals, to judge trend also.

A box plot is constructed by plotting points $(0, u_i)$, along a line and marking the median and quartiles. The positions of these relative to the end points 0 and 1 will reveal any skewing of the distribution.

Pearson's chi-squared statistic (see **B.3.2.5**) may be used to test the distribution of the u_i by partitioning the interval [0,1] into k intervals of equal length $1/k$, and using the fact that the expected number of u_i in each interval is $E_i = (c - s)/k$.

The procedure to construct a probability plot or 'u-plot' is as follows.

a) Compute the u_i $(s + 1 \leq i \leq c)$ as above.

b) Rank the u_i in ascending order. Denote the sorted sequence u_j, $j = 1, ..., (c - s)$.

c) Plot the points $(u_j, j/(c - s + 1))$, i.e. as each u_j is plotted against the horizontal axis, a constant step $1/(c - s + 1)$, is taken up the vertical axis.

d) Locate the point of maximum deviation of the plot from the line of unit slope from (0,0) to (1,1), and measure the vertical distance of the plot from the line at that point. This is the Kolmogorov-Smirnoff (K-S) distance (see **B.3.2.4**). (Note that the u-plot procedure destroys the time-ordering of the points.)

e) The deviation of the u_i from U(0,1) is shown by the deviation of the plot from the line of unit slope. If the point of maximum vertical distance lies above the line, then the $\hat{F}_i(t)$ are biased optimistically. If it lies below the line, the bias is pessimistic. The K-S distance is a measure of the significance of the deviation (the degree of confidence that it is not due to random variation).

A typical u-plot is illustrated in figure 11.

C.2.3 *Assessment of trend in predictions*

Any trend in the u_i with increasing i indicates that the prediction system is not adequately capturing some trend in the observed t_i, e.g. degree of reliability growth. This is most easily detected as a trend in the x_i or y_i.

Tests for trend such as the Laplace test (see **B.3.2.2**) may be applied to the x_i. Since the sequence of x_i should be HPP, the hazard intensity of the underlying process may be estimated in order to assess if it is constant over time.

Another technique is to construct a probability plot or 'y-plot' of points $(y_i, i/(c - s))$ following the same procedure as for the u-plot. (Note that there are only $(c - s - 1)$ y_i values.)

In this case, deviation from the line of unit slope indicates departure of trend in the predictor distributions $\hat{F}_i(t)$ from trend in the observed t_i. Again, its significance is measured by the K-S distance.

C.2.4 *Assessing other types of inaccuracy*

C.2.4.1 *Other causes of inaccuracy*

Predictive inaccuracy may arise from other causes than bias or failure to capture trend. One possibility is 'noise', i.e. predictions may be unstable in i (e.g. pessimistic in one case, optimistic in the next) so that the predictor CDF $\hat{F}_i(t)$ varies widely around the 'true' CDF $F_i(t)$, but in such a way that (although each individual prediction is poor) predictions are unbiased on average, and follow overall trend.

Another possibility is that accuracy varies for different parts of the distribution, e.g. predictions may be pessimistically biased in the left tail, fairly accurate around the mean, and optimistically biased in the right tail. (This could be observed in the u-plot, which would lie below the line of unit slope to the left, cross it around the mid-point, and lie above it on the right.)

Other subtle variations in accuracy may be found.

Inaccuracy due to excessive variability in predictions (i.e. variability that does not model a real underlying variability in the data) can be detected by *median variability* and *rate variability* tests. Differences in the accuracy of two prediction systems due to any cause may be detected by trend in *prequential likelihood*. These tests are described in **C.2.4.2** and **C.2.4.3**.

C.2.4.2 *Variability tests*

Variability of the predicted MeTTF may be assessed by plotting it (together with the upper and lower quartiles of predicted TTF) against failure number i or accumulated operating time x. Any instability of the estimates should be obvious to the eye. Where 'time to failure' data is available, the actual t_i may be plotted on the same diagram. Around 50 % of the actual points should lie between the upper and lower quartiles.

Variability of the predicted ROCOF may be detected in similar plots. In this case, the values to be plotted might include the empirical failure rate estimated by $1/t_i$, c_i/u_i, or some suitable rolling average of these. (Here, c_i/u_i is used with the meaning of **B.3**, i.e. u_i is the operating time in period i.)

The variability of any sequence of predicted values $\hat{q}_j$ (over j data points, after s initial observations) may be measured by the statistic:

$$\text{vary}\{\hat{q}_{s+1}, ..., \hat{q}_j\} = \sum_{i=s+1}^{j-1} \left| \frac{\hat{q}_{i+1} - \hat{q}_i}{\hat{q}_i} \right|$$

This represents the variability of the predictions, but provides no level of significance. In addition, it does not compare the predictions with the actual values. (It is often observed that actual TTF is very unstable, and fluctuates widely around the predicted median, with many points outside the predicted inner quartiles.) The Braun statistic (see **C.2.4.4**) and relative error measures may also reveal inappropriate variability.

C.2.4.3 *Prequential likelihood*

Prequential likelihood makes use of the predictor PDF $\hat{f}_i(t)$, where the predictor CDF:

$$\hat{F}_i(t) = \int_0^t \hat{f}_i(t)\mathrm{d}t$$

It depends upon the fact that, if the predictor PDF $\hat{f}_i(t)$ is 'close to' the 'true' PDF $f_i(t)$, then the values $\hat{f}_i(t_i)$ and $f_i(t_i)$, for the realization t_i will be close, i.e. the ratio $\hat{f}_i(t_i) / \hat{f}_i(t_i)$ will tend to lie close to 1. The prequential likelihood (PL) of a prediction system A is defined as:

$$\text{pl}_\text{A}(s,c) = \prod_{i=s+1}^{c} \text{x}\, \hat{f}_i^\text{A}(t_i)$$

where c is the number of observations and s is the minimum number required for prediction to begin (as in **C.2.1** above). (The superscript A on the predictor PDF $\hat{f}_i^A(t)$ indicates that it is derived from prediction system A.)

The prequential likelihood ratio (PLR) for two prediction systems A and B is defined as follows.

$$\text{plr}_{A,B}(s,c) = \text{pl}_A(s,c) / \text{pl}_B(s,c)$$

If $\text{pl}_T(s,c)$ denotes the PL for the 'true' PDFs $f_i(t)$, then the trend in $\text{plr}_{A,T}(s,c)$ as c increases could indicate how close A is to the truth, but the $f_i(t)$ are unknown. However, it can be shown that, if $\text{plr}_{A,B}(s,c) \to 0$ as $c \to \infty$, then prediction system A is discredited as a predictor compared to prediction system B (and conversely, B is discredited if $\text{plr}_{A,B}(s,c) \to \infty$). The trend in $\text{plr}_{A,B}(s,i)$ as i increases from $(s + 1)$ to c can therefore be used to compare the predictive accuracy of A and B. (In practice, the trend is usually very marked. Due to the sizes of the quantities involved, it is usual to calculate the logarithm of the PLR.)

If more than two prediction systems are to be compared, then one system can be chosen to act as a reference, and the trends in the PLRs for the others can be compared relative to it.

C.2.4.4 *Other measures of accuracy*

The accuracy of any predicted value $\hat{q}$ may be measured by the *relative error* as follows.

$$\text{rele}\{\hat{q}\} = (\hat{E}\{q\} - q)/q$$

where q is the actual value eventually observed.

This may be applied to predictions of TTF, $c(x)$, etc. Plots of relative error may be used to show trend in the accuracy of a sequence of predictions, and comparison of the relative error in predictions from two prediction systems may be used to compare them, but it does not provide an assessment of the significance of the error.

The Braun statistic may be used to assess the accuracy of a sequence of predicted mean times to failure $\hat{E}\{T_i\}$. It is defined as follows.

$$\text{braun}\{\hat{E}\{T_s+1\}, ..., \hat{E}\{T_i\}\} = \frac{(i-s-1)\sum_{j=s+1}^{i}(t_j - \hat{E}\{T_j\})^2}{(i-s-2)\sum_{j=s+1}^{i}(t_j - \bar{t})^2}$$

Here, s is the number of initial TTFs used as the basis for the first prediction (as previously), t_j is the jth inter-failure time, and $\bar{t}$ is the average of all the observed inter-failure times up to and including t_i. The smaller the value of the Braun statistic, the better the prediction system is judged to be. In particular, a value greater than 1 would indicate a very poor prediction system.

NOTE. The average of the observed inter-failure times is not a particularly meaningful statistic where reliability growth is occurring. However, the term

$$\sum_{j=s+1}^{i}(t_j - \bar{t})^2$$

in which it occurs may be regarded as a normalizing constant, and the Braun statistic may be used to compare prediction systems, rather than as an absolute indicator of accuracy. (The prediction system with the lower Braun value is preferred.)

For 'failure count' predictions, the following form of the Braun statistic may be used.

$$\text{braun}\{\hat{E}\{K_{s+1}\}, ..., \hat{E}\{K_p\}\} = \frac{\sum_{j=s+1}^{p}(k_j - \hat{E}\{K_j\})^2 u_j}{\sum_{j=s+1}^{p}(k_j - \bar{k})^2 u_j}$$

Here, p is the number of periods, and s is the number of periods used as the starting point for the first prediction. u_j is the amount of operating time in the jth period, and $\bar{k}$ is the average of the counts k_j of failures in each period.

C.3 Recalibration of predictions

If the inaccuracy of a prediction system on a given data set is due mainly to bias (i.e. if it yields a 'poor' u-plot, but a 'good' y-plot), then its predictions may be improved by recalibration to reduce the bias. This is referred to as 'adaptation' of the prediction system or model to the data set, and a procedure for doing this in the case of 'one step ahead' prediction on a 'time to failure' data set is described here.

As before, denote the 'true' and predictor distributions of T_i respectively by $F_i(t)$ and $\hat{F}_i(t)$. The problem of adaptation is solved if a function G_i:(0,1) $\to$ (0,1) can be found for each i such that $F_i(t) = G_i(\hat{F}_i(t))$, and which fulfils two conditions as follows.

a) In order for $G_i(\hat{F}_i(t))$ to be a genuine predictor, $G_i(.)$, like $\hat{F}_i(.)$, should be derived solely from $\{t_1, ... , t_{i-1}\}$ and not depend on any later observations;
b) In order for $G_i(.)$to be an effective adaptor, the bias in $\hat{F}_i(.)$ should be similar to that observed in $\hat{F}_{i-1}(.)$ and earlier predictor CDFs, i.e. the extent and direction of the bias should be 'stationary'.

This is an ideal solution only, since $F_i(t)$ is inherently unknowable. However, the u-plot (see **C.2.2**) is defined for $j = s + 1, ..., c$ as a function:

$$G_c^U:\{\hat{F}_j(t_j)\}\rightarrow\{(j - s) / (c - s + 1)\}$$

where (as before) c is the total number of observations and s is the minimum number needed for the first prediction. The u-plot is the sample CDF of the $\{u_s + 1, ..., u_c\}$, and maps $\hat{F}_j(t_j)$ onto the U(0,1) realization corresponding to $F_j(t_j)$. If its domain included all $\hat{F}_j(t)$ instead of just a discrete subset, it could approximate to the adaptor function $G_i(.)$ sought.

The 'adaptive modelling' procedure to estimate $T_i(i > s)$ is therefore as follows.

1) The predictor CDFs $\{\hat{F}_{s+1}(.), ..., \hat{F}_i-1(.)\}$ are derived from the prediction system.

2) The $\{t_{s+1}, ... , t_{i-1}\}$, are substituted into the predictor CDFs to give the $(i - s - 1)$ values u_j and the u-plot is constructed.

3) The corresponding y-plot is constructed and trend capture is assessed.

4) If the u-plot shows significant bias but the y-plot shows reasonable capture of trend, then adaptation is likely to be beneficial and may proceed.

5) To interpolate between the u_j values $\{\hat{F}_{s+1}(t_{s+1}), .., \hat{F}_{i-1}(t_{i-1})\}$, successive points $(u_j, j/(i - s))$, $(u_{j+1}, (j + 1)/(i - s))$, are connected on the u-plot. (0,0) is connected to $(u_{s+1}, (s + 1)/(i - s))$ and $(u_{i-1}, (i - 1)/(i - s))$ is connected to (1,1). This defines a function $G_i^C(.)$ which is continuous, but consists of a sequence of straight-line segments and so has a discontinuous first derivative.

6) $G_i^C(.)$ is smoothed by fitting a spline curve to give a function $G_i^*(.)$ with a continuous first derivative $g_i^*(.)$. The adapted predictor CDF for T_i is then $\hat{F}_i^*(t) = G_i^*(\hat{F}_i(t))$ and its corresponding PDF is $\hat{f}_i^*(t) = g_i^*(\hat{F}_i(t))\hat{f}_i(t)$.

The adaptive procedure applied to any existing prediction system A provides a new prediction system A* whose accuracy can be assessed by the same techniques as used on the original, in particular u-plots, y-plots, and prequential likelihood. (The reason for not using $G_i^C(.)$ as the adaptor function is that its discontinuous first derivative means that PL cannot be used, since it makes use of the predictor PDF.)

Generally, on a set of adapted predictions $\{\hat{F}_i^*(t)\}$ and a set of corresponding observations $\{t_i\}$ $(i = s + 1, ..., c)$, the u*-plot shows minimal bias, the y*- plot is no worse than the original y-plot, and the PLR $\text{plr}_{A^*A}(s,c)$ discredits the original prediction system A compared with the adapted system A*. It is often found that predictions from several prediction systems applied to the same set of observations differ widely. In such cases, adaptation is also found to bring the differing estimates into closer agreement.

C.4 Combination of predictions

Another way of improving predictions is to combine the results of applying several different prediction systems. A linear combination of m prediction systems at each step j in 'one step ahead' predictions may be defined as follows.

$$\hat{F}_j^C(t) = \sum_{r=1}^{m} w_j^r \hat{F}_j^r(t) \text{ where } \sum_{r=1}^{m} w_j^r = 1$$

w_j^r is the weighting factor for prediction system $r\{1, ..., m\}$at step j. One way in which these weights may be chosen is to use prequential likelihood ratios as follows.

$$w_j^r = \text{plr}_{1,r}(s, j-1) / \sum_{k=1}^{m} \text{plr}_{1,k}\ (s, j - 1)$$

It is assumed that prediction system 1 is used as the reference against which the PLR of each other system is calculated. Note that the combined predictor distribution is a true predictor, since it depends only upon information obtained up to failure $j-1$.

Such combinations of prediction systems have been found to be give improved results compared to those of the individual raw predictors.

C.5 Discrete predictions

Some of the methods of assessing predictive accuracy are applicable to discrete predictions, i.e. predictions of 'failure count' in future intervals of given length. Examples are the use of the Pearson (see **B.3.2.5**) and Braun (see **C.2.4.4**) statistics, variability tests (see **C.2.4.2**) and measures of relative error (see **C.2.4.4**).

Other techniques are restricted to 'time to failure' predictions, since they depend upon the continuous nature of the predictor distribution. In particular, the u- and y-statistics (see **C.2.1**) are limited in this way, and the PLR (see **C.2.4.3**) may not perform well if the predictions and realizations are each of a small number of failures (e.g. 2 or 3 per interval).

This also restricts the use of recalibration (see **C.3**) or combination (see **C.4**) of predictions which depend on the use of the u-plot as a recalibration function or of the PLR as a weighting factor.

Extensions to the u-plot and y-plot techniques to cope with discrete predictions have been devised, but a description is beyond the scope of this annex.

C.6 Long-term predictions

The methods in this annex (particularly u-plots, y-plots and PLR) have been described with reference to 'one step ahead' prediction. They can be extended in a fairly obvious way to longer-term predictions. The most useful assessment of a long-term prediction is probably the relative error in the predicted failure count. Ideally, this should take account of confidence limits expressed in terms of percentiles, rather than a simple point estimate.

Annex D (informative)
Bibliography

[1] DANIELS, B.K. Error, fault and failure in software production. *In*: *Proc. Human Factors in the Process Control Centre*. Manchester, UK: Institute of Chemical Engineering, April 1983.

[2] LITTLEWOOD B. and L. STRIGINI. The risks of software. *Scientific American*, November 1992, **267**(5), 38–43.

[3] FENTON N.E. and S.L. PFLEEGER. *Software metrics: a rigorous and practical approach*, 2nd edition. London: International Thomson Computer Press, 1996. ISBN 1 85032 275 9.

[4] MELLOR, P. Failures, faults and changes in software dependability measurement. *Information and software technology*, 1992, **34**(10), 640–654.

[5] EUROCAE/ED – 12B: *Software considerations in airborne systems and equipment certification*. European Organization for Civil Aviation Electronics(EUROCAE), jointly with Requirements and Technical Concepts for Aviation (RTCA), January 1993. (Referred to in the United States as RTCA/DO-178B.)

[6] HEALTH AND SAFETY EXECUTIVE (HSE). *Programmable electronic systems in safety related applications. Part 1: An introductory guide. Part 2: General technical guidelines*. London: The Stationery Office, 1987.

[7] GILB, T. Design by objectives. Amsterdam: North- Holland, 1985.

[8] KITCHENHAM, B.A., A.P. KITCHENHAM and J.P. FELLOWS. The effects of inspections on software quality and productivity. *ICL Technical J.*, May 1986, **5**(1), 112–122.

[9] FAGAN, M.E. Design and code inspections to reduce errors in program development. *IBM Systems J.*, 1976, **15**(3), 182–211.

[10] REMUS, H and S. ZILLES. Prediction and management of program quality. *In*: *Proc. 4th Int. Conf. on Software Engineering*. New York: IEEE Computer Society Press, 1979, pp. 341–350.

[11] GAFFNEY, J.E. Estimating the number of faults in code. *IEEE Trans. Software Engineering*, 1984, **SE-10**(4), 459–465.

[12] KITCHENHAM, B.A. Towards a constructive quality model. *IEE Software Engineering J.*, 1987, **2**(4), 105–113.

[13] KITCHENHAM, B.A. Metrics in practice. *In*: A. BENDELL and P. MELLOR, eds. *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986, pp. 131–144 and pp. 253–255.

[14] JOE, H and N. REID. Estimating the number of faults in a system. *J. American Statistical Association*, 'Theory and Methods' series, 1985, **80**(389), 222–226.

[15] AMJ 25.1309. System design and analysis. *Advisory material to Joint Airworthiness Requirements*, Part 25, paragraph 1309 (JAR-25.1309), Joint Airworthiness Authority (JAA), 1990. Available from Civil Aviation Authority, Gatwick Airport, UK.

[16] HALSTEAD, M.H. *Elements of software science*. Amsterdam: Elsevier North-Holland, 1977.

[17] HAMER, P.G. and G.D. FREWIN. *M.H. Halstead's software science – a critical examination. In*: *Proc. 6th Int. Conf. on Software Engineering*. New York: IEEE Computer Society Press, 1982, pp. 197–206.

[18] LIPOW, M. and T.A. THAYER. Prediction of software failures. *In*: *Proc. Annu. Reliability and Maintainability Symposium*. New York, IEEE Computer Society Press, 1977, pp. 489–494.

[19] AKIYAMA, F: An example of software system debugging, *Information processing*, 1971, **71**, 353–379.

[20] McCABE, T.J. A complexity metric, *IEEE trans. software engineering*, December 1976, **SE-2**(4), 308–320.

[21] KITCHENHAM, B.A. Measures of programming complexity. *ICL Technical J.*, 1981, **2**(3), 298–316.

[22] WALTERS, G.F. and J.A. McCALL. The development of metrics for software R and M, *In*: *Proc. Annu. Reliability and Maintainability Symposium*. New York: IEEE Computer Society Press, 1978, pp. 79–85.

[23] BOEHM, B.W., J. R. BROWN, H. KASPAR, M. LIPOW, G. J. McLEOD and M.J. MERRITT. *Characteristics of software quality*. TRW series on 'Software technology'. Amsterdam: North Holland, 1978.

[24] McCALL, J.A., P.K. RICHARDS and G.F. WALTERS. *Factors in software quality*. Rome Air Development Center reports NTIS AD/A-049 014 (Vol. I), 015 (Vol. II), and 055 (Vol. III). Griffiss Air Force Base, NY: Rome Air Development Center, 1977, pp. 308–320.

[25] FARR, W. Software reliability modeling survey. *In*: M.R. LYU, ed. *Handbook of software reliability engineering*. New York: IEEE Computer Society Press/McGrawHill, 1996, pp. 71–115. ISBN 0 07 039400 8.

[26] MILLER, D.R. and A. SOFER. A non parametric approach to software reliability using complete monotonicity. *In*: *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986, pp. 183–195 and pp. 257–258.

[27] BROCKLEHURST, S. A non parametric approach to software reliability modelling. Report delivered by ESPRIT project *Predictably dependable computer systems* (BRA 30920), October 1989.

[28] BOX, G.E.P. and G.M. Jenkins. *Time series analysis: forecasting and control*. San Francisco, CA: Holden-Day, 1976.

[29] DALE, C.J. and L.N. HARRIS. Approaches to software reliability prediction. *In*: *Proc. Annu. Reliability and Maintainability Symposium*. New York: IEEE Computer Society Press, 1982, pp. 167–175.

[30] DALE, C.J., L.N. HARRIS, P.D.T. O'CONNOR and G. RZEVSKI. *Reliability aspects of microprocessor systems*. British Aerospace Dynamics Group, Stevenage Division, Report no. ST25358, (Department of Industry report no. T816164), 1981.

[31] SINGPURWALLA, N.D. Time series analysis of failure data. *In*: *Proc. Annu. Reliability and Maintainability Symposium*. New York: IEEE Computer Society Press, 1978, pp. 107–112.

[32] SOYER, R. Application of time series models to software reliability analysis. *In*: *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986, pp. 197–207 and pp. 258–259.

[33] GRAY, C.T. A framework for software reliability modelling. *In*: *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986 , pp. 81–94 and pp. 249–250.

[34] GOEL, A.L. and K. OKUMOTO. An imperfect debugging model for reliability and other quantative measures of software systems. *In*: *Bayesian software prediction models*, Rome Air Development Center report RADC-TR-78-155, Issue 5, Vol. 1, Griffiss Air Force Base, NY: Rome Air Development Center, 1978.

[35] OKUMOTO, K. A statistical method for software quality control. *IEEE trans. software engineering*, 1985, **SE-11**(12), 1424–1430.

[36] NELDER, J.A. and R.A. MEAD. A simplex method for function minimization. *The Computer Journal*, 1965, **7**, 308–313.

[37] CHAN P.Y. *Software reliability prediction*, PhD thesis, City University, London, 1986.

[38] MOEK, G. *Maximum likelihood estimation of software reliability model parameters*. Memorandum IW-86-006L, National Aerospace Laboratory (NLR), Informatics Division, Amsterdam, Netherlands, 1982.

[39] JELINSKI, Z. and P.B. MORANDA. Software reliability research. *In*: W. FREIBERGER, ed. *Statistical computer performance evaluation*. New York: Academic Press, 1972, pp. 465–484.

[40] ADAMS, E.N. *Minimizing cost impact of software defects*. IBM Research Division report no. RC 8228 (#35669), IBM Thomas J. Watson Resarch Centre, New York, November 1980.

[41] SHOOMAN, M.L. Operational testing and software reliability estimation during program developments. *In*: *Record of 1973 IEEE Symposium on Computer Software Reliability*. New York: IEEE Computer Society Press, 1973, pp. 51–57.

[42] SCHICK, G.J. and R.W. WOLVERTON. An analysis of competing software reliability models. *IEEE trans. software engineering*, 1978, **SE-4**(2), 104–120.

[43] MORANDA, P.B. Prediction of software reliability during debugging. *In*: *Proc. Annu. Reliability and Maintainability Symposium*, Washington, DC, 1975, pp. 327–322.

[44] SCHAFER, R.E. J.E. ANGUS, J.F. ALTER and S.E. EMOTO. *Validation of software reliability models*. USAF report RADC-TR-79-147, 1979.

[45] ANGUS, J.E., R.E. SCHAFER and A.N. SUKERT. Software reliability model validation. *In*: *Proc. Annu. Reliability and Maintainability Symposium*. New York: IEEE Computer Society Press, 1980, 191–199.

[46] BROOKS, W.D. and R.W. MOTLEY. *Analysis of discrete software reliability models*. Rome Air Development Center report no. RADC-TR-80-84. Griffiss Air Force Base, NY: Rome Air Development Center, April 1980.

[47] LITTLEWOOD, B. Stochastic reliability growth: a model for fault removal in computer programs and hardware designs, *IEEE trans. reliability*, 1981, **R- 30**(4), 313–320.

[48] WEISS, H.K. Estimation of reliability growth in a complex system with a Poisson-type failure. *Operations research*, 1956, **4**(5), 532–545.

[49] THAYER, T.A., G.R. CRAIG and L.E. FREY. *Software reliability study*. Rome Air Development Center report no. RADC-TR-76-238. Griffiss Air Force Base, NY: Rome Air Development Center, 1976.

[50] RAMAMOORTHY, C.V. and F.B. BASTANI. Modelling of the software reliability growth process, *In*: *COMPAC '80*, 1980, pp. 161–169.

[51] MUSA, J.D., A. IANNINO and K. OKUMOTO. *Software reliability measurement, prediction, application*. New York: McGraw-Hill, 1986.

[52] GOEL, A.L. and K. OKUMOTO. Time dependent error detection rate model for software reliability and other performance measures. *IEEE trans. reliability*, 1979, **R-28**(3), 206–211.

[53] MUSA, J.D. A theory of software reliability and its application. *IEEE trans. software engineering*. 1975, **SE-1**(3), 312–327.

[54] MUSA, J.D. and K. OKUMOTO. Application of basic and logarithmic Poisson execution time models in software reliability measurement. *In*: J.K. SKWIRZYNSKI, ed. *Proc. NATO Advanced Study Institute (Durham, 1985)*, NATO ASI Series F, 'Computer and system sciences', Vol. 22. Berlin: Springer-Verlag, 1986.

[55] DONELSON, J. *Duane's reliability growth model as a non-homogeneous Poisson process*. Institute for Defence Analyses report no. P-1162, 1975, Program Analysis Division, 400 Army-Navy Drive, Arlington, VA 22202, USA.

[56] RUSHFORTH, C.K., F.L. STAFFANSON and A.E. CRAWFORD. *Software reliability estimation under conditions of incomplete information*. Rome Air Development Center report no. RADCTR-79-230. Griffiss Air Force Base, NY: Rome Air Development Center, 1979.

[57] SCHNEIDEWIND, N.F. Analysis of error processes in computer software. *In*: *Proc. Int. Conf. on Reliable Software*, 1975, pp. 336–346.

[58] LANGBERG, N. and N.D. SINGPURWALLAH. *A unification of some software reliability models via the Bayesian approach*, George Washington University technical memo TM-66571, 1981, George Washington University, Washington, DC.

[59] MILLER, D.R. Exponential order statistic models of software reliability growth. *IEEE trans. software engineering*, 1986, **SE-12**(1), 12–24.

[60] DUANE, J.T. Learning curve approach to reliability monotoring. *IEEE Trans. Aerospace*, April 1964, **2**, 563–566.

[61] LITTLEWOOD, B. and J.L. VERRALL. A Bayesian reliability growth model for computer software. *J. Royal Statistical Society*, 1973, **C-22**(3), 332–346.

[62] KEILLER, P.A., B. LITTLEWOOD, D.R. MILLER and A. SOFER. On the quality of software reliability predictions. *In*: J.K. SKWIRZYNSKI, ed. *Proc. NATO Advanced Study Institute (Norwich)*. Berlin: Springer-Verlag, 1983, pp. 441–460.

[63] COX, D.R. Regression models and life tables. *J. Royal Statistical Society*, 1972, **B34**(2), 187–220.

[64] NAGEL, P.M. and J.A. SKRIVAN. *Software reliability: repetitive run experimentation and modeling*. Boeing report no. BCS-40366. Seattle, WA: Boeing Computer Services Company, 1981.

[65] WIGHTMAN, D.W. and A. BENDELL. Proportional hazards modelling of software failure data. *In*: *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986, pp. 229–242 and pp. 261–263.

[66] IYER, R.K. and D.J. ROSETTI. Effect of system workload on operating reliability: a study on IBM 3081. *IEEE Transactions on Software Engineering*, 1985, **SE-11**(12), 1438–1448.

[67] DALE, C.J. and L.N. HARRIS. *Software reliability evaluation methods*. British Aerospace Dynamics Group, Stevenage Division, report no. ST26750, 1982.

[68] DALE, C.J. Software reliability models. *In*: *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986, pp. 31–44 and pp. 244–247.

[69] NELSON, E.C. Estimating software reliability from test data. *Microelectronic Reliability*, **17**(1), 67–74.

[70] DURAN, J.W. and J.J. WIORKOWSKI. Quantifying software reliability by sampling. *IEEE Trans. Reliability*, 1980, **R-29**(2), 141–144.

[71] RAMAMOORTHY, C.V. and F.B. BASTANI. Software reliability – status and perspectives. *IEEE trans. software engineering*, July 1982, **SE-8**(4), 354–371.

[72] FELLER, W. *An introduction to probability theory and its applications*, Vol. 1, 2nd edition. London: Wiley, 1957.

[73] CHAPMAN, D.G. Some properties of the hypergeometric distribution. *University of California Publications in Statistics*, 1951, **1**(7), 131–160.

[74] MILLS, H.D. *On the statistical validation of computer programs*, FSC-72-6015, IBM Federal Systems Division, Gaithersburg, MD, 1972.

[75] RUDNER, B. *Seeding/tagging estimations of software errors: models and estimates*. RADC-TR-77-15, Rome Air Development Center, A036655. Griffiss Air Force Base, NY: Rome Air Development Center, 1977.

[76] TRIVEDI, A.K. and M. SHOOMAN. *Computer software reliability: many-state Markov modeling techniques*. Rome Air Development Center report no. RADC-TR-75-169. Griffiss Air Force Base, NY: Rome Air Development Center, 1975

[77] ABDEL-GHALY, A.A. *Analysis of predictive quality of software reliability models*. PhD Thesis, City University, London, 1986.

[78] CHAN, P.Y. 'Adaptive models', *Pergamon Infotech state of the art report on 'Software reliability'*. Maidenhead, Berks., UK: Pergamon Infotech, 1986, pp. 3–29 and pp. 243–244.

[79] LITTLEWOOD, B. A.A. ABDEL-GHALY and P.Y. CHAN. Tools for the analysis of the accuracy of software reliability predictions. *In*: J.K. SKWIRZYNSKI ed. *Proc. NATO Advanced Study Institute (Durham, 1985)*, NATO ASI Series F, 'Computer and system sciences', Vol. 22. Berlin: Springer-Verlag, 1986, pp. 299–335.

[80] WRIGHT, D.R. Recalibrated prediction of some software failure count sequences. *In*: *Predictably dependable computer systems (PD CS)*, ESPRIT Project 6362, first year report, September 1993, pp. 337–365.

[81] DAWID, A.P. Statistical theory: the prequential approach to statistical theory. *J. Royal Statistical Society*, 1984, **A147**, 278–292.

[82] BROCKLEHURST, S. and B. LITTLEWOOD. Techniques for prediction analysis and recalibration. *In*: M.R. LYU, ed. *Handbook of software reliability engineering.* New York: IEEE Computer Society Press/McGraw-Hill, 1996, pp. 119–164. ISBN 0 07 039400 8.

[83] LU, M., S. BROCKLEHURST and B. LITTLEWOOD. Combination of predictions from different software reliability growth models. *J. computer and software engineering*, 1993, **1**(4), 303–323.

[84] LITTLEWOOD, B. Software reliability model for modular program structure. *IEEE trans. reliability.* 1979, **R-28**(3), 241–246.

[85] CHEUNG, R.C. A user orientated software reliability model. *IEEE trans. software engineering*, March 1980, **SE-6**(2), 118–125.

[86] KANOUN, K., M. KAÂNICHE, C. BEOUNES, J.C. LAPRIE and J. ARLAT. Reliability growth of fault-tolerant software. *In*: *Predictably dependable computer systems (PD CS2)*, ESPRIT Project 3092, second year report, Vol. 2, chapter 2, May 1991.

[87] LAPRIE, J.C. and K. KANOUN. X-ware reliability and availability modelling. *IEEE trans. software engineering*, February 1992, **18**(2), 130–147.

[88] LAPRIE, J.C. and K. KANOUN. Software reliability and system reliability. *In*: M.R. LYU ed. *Handbook of software reliability engineering.* New York: IEEE Computer Society Press/McGraw-Hill, 1996, pp. 27–68. ISBN 0 07 039400 8.

[89] LITTLEWOOD, B. and L. STRIGINI. Validation of ultra-high dependability for software-based systems. *Communications Assoc. Computing Machinery*, 1993, **36**(11), 69–80.

[90] LUTZ, R.R. *Targeting safety-related errors during requirements analysis*. *In*: D. NOTKEN, ed. *Proc. 1st ACM SIGSOFT Symp., ACM Press Software Engineering Notes*, 1993, **18**(5), 99–105.

[91] KNIGHT, J.C. and N.G. LEVESON. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 1986, **SE-12**(1), 96–109.

[92] LITTLEWOOD, B. and D.R. MILLER. A conceptual model of multi-version software. *Proc. 17th Annu. Int. Symp. fault-tolerant computing*, Pittsburgh, PA, July 1987. Silver Spring, MD: IEEE Computer Society Press, pp. 150–155.

[93] ECKHARDT, D.E and D.L. LEE. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, 1985, **SE-11**(12), 1511–1517.

[94] MILLER, D.R. Making statistical inferences about software reliability. Invited paper, *Joint Statistical Meetings*, Chicago, 1986. Also published in: *Alvey Software Reliability and Metrics Club newsletter*, December 1986, (4), 3–22, Centre for Software Reliability, City University, London.

List of references (see clause 2)

Normative references

BSI publications

BRITISH STANDARDS INSTITUTION, London

BS 4778	*Quality vocabulary*
BS 4778 : Part 3	*Availability, reliability and maintainability terms*
BS 4778 : Section 3.2 : 1991	*Glossary of international terms*
BS 5760	*Reliability of systems, equipment and components*
BS 5760 : Part 2	*Guide to assessment of reliability*
BS 6488: 1984 (1992)	*Code of practice for configuration management of computer-based systems*
BS EN ISO 8402 : 1995	*Quality management and quality assurance. Vocabulary*

BSI — British Standards Institution

BSI is the independent national body responsible for preparing British Standards. It presents the UK view on standards in Europe and at the international level. It is incorporated by Royal Charter.

Revisions

British Standards are updated by amendment or revision. Users of British Standards should make sure that they possess the latest amendments or editions.

It is the constant aim of BSI to improve the quality of our products and services. We would be grateful if anyone finding an inaccuracy or ambiguity while using this British Standard would inform the Secretary of the technical committee responsible, the identity of which can be found on the inside front cover. Tel: 020 8996 9000. Fax: 020 8996 7400.

BSI offers members an individual updating service called PLUS which ensures that subscribers automatically receive the latest editions of standards.

Buying standards

Orders for all BSI, international and foreign standards publications should be addressed to Customer Services. Tel: 020 8996 9001. Fax: 020 8996 7001.

In response to orders for international standards, it is BSI policy to supply the BSI implementation of those that have been published as British Standards, unless otherwise requested.

Information on standards

BSI provides a wide range of information on national, European and international standards through its Library and its Technical Help to Exporters Service. Various BSI electronic information services are also available which give details on all its products and services. Contact the Information Centre. Tel: 020 8996 7111. Fax: 020 8996 7048.

Subscribing members of BSI are kept up to date with standards developments and receive substantial discounts on the purchase price of standards. For details of these and other benefits contact Membership Administration. Tel: 020 8996 7002. Fax: 020 8996 7001.

Copyright

BSI
389 Chiswick High Road
London
W4 4AL