ESA/ESTEC Study Contract	Doc. Id	PASCON/WO6-CCN2/TN12
N. 10662/93/NL/NB WO6-CCN	Issue	3.1
	Date	8 October 1997

Analysis, Specification and Verification/Validation of Software Product Assurance Process and Product Metrics for Reliability and Safety Critical Software

Guidelines for Statistical Testing

Written by:	B. Littlewood L. Strigini Centre for Software Reliability, City University, London, United Kingdom
Reviewed by:	P. Panaroni / Intecs Sistemi P. Popov / Centre for Software Reliability, City University

Table of Contents

Appendices A and B (pp. 78-85) have been omitted from this copy

1. Introduction

1.1 Scope

This Technical Note (TN) is the result of WP A12 of the project "Analysis, Specification and Verification/Validation of Software Product Assurance Process and Product Metrics for Reliability and Safety Critical Software". The project is identified as Work Order No. 6 CCN-0 I of ESTEC Contract No. 10662/93/NL/NB

This Technical Note is identified as TN 12 and has been prepared by the Centre for Software Reliability, City University, under the supervision of Intecs Sistemi in Work Package A12.

An electronic version of this report is available, upon request, in Postscript format. Comments on the usability and contents of this document are welcome and should be sent to:

Lorenzo Strigini
Centre for Software Reliability, City University, Northampton Square, London EC1V OHB, UK
Fax +44 171 477 8585; E-mail: strigini@csr.city.ac.uk

1.2 Purpose

This document provides an introduction to statistical testing for the use of ESA suppliers.

Statistical testing of software is here defined as testing in which the test cases are produced by a random process meant to produce different test cases with the same probabilities with which they would arise in actual use of the software. Statistical testing of software has these main advantages:

- for the purpose of reliability assessment and product acceptance, it supports directly estimates of reliability, and thus decisions on whether the software is ready for delivery or for use in a specific system. This feature is unique to statistical testing;

- for the purpose of improving the software, it tends to discover defects which would cause failures with the higher frequencies before those that would cause less frequent failures, thus focusing correction efforts in the most cost-effective way and delivering better software for a given debugging effort. Statistical testing has been reported to achieve dramatic improvements;

- from the point of view of costs, it facilitates the automation of the test process, thus allowing more testing at acceptable cost than manual testing would allow.

This document explains the basic theory underlying statistical testing and provides guidance for its application. The material is organised to facilitate use both as an introduction for software engineers who are new to this approach to testing, and as a reference source during application.

Statistical testing is applicable to practically all kinds of software, so this document is not markedly specialised for space applications, though the examples are mostly space-related and the discussion of the software lifecycle is meant to apply to common practice among ESA suppliers.

Where practical, material that is useful for quick reference use is set out in boxes as shown below.

Quick reference material

Boxes like this contain quick reference material (checklists and outlines of mathematical procedures).

2. Applicable and reference documents

2.1 Applicable documents

[SOW] Statement of work ref. QP/95G-429/LW/P1555, 14/7/95

]Proposal] PASCON Work order 6, Work Plan for Phase 2 (TN2), ESTEC Contract No. 10062/93/NL/NB, Report No. CLB/6620DVD, Revision No. 05

2.2 Reference documents

[Abdel-Ghaly 1986] A. A. Abdel-Ghaly, P. Y. Chan, B. Littlewood, "Evaluation of Competing Software Reliability Predictions", *IEEE Transactions on Software Engineering*, SE-12 (9), pp.950-67, 1986.

[Adams 1984] E. N. Adams, "Optimizing preventive maintenance of software products", *IBM J. of Research and Development*, 28 (1), pp.2-14, 1984.

[Aitchison 1975] J. Aitchison, I. R. Dunsmore, *Statisitical Prediction Analysis,* p., Cambridge University Press, Cambridge, 1975.

[Avritzer 1995] A. Avritzer, E. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software", *IEEE TSE*, 21 (9), pp.705-16, 1995.

[Brocklehurst 1990] S. Brocklehurst, P. Y. Chan, B. Littlewood, J. Snell, "Recalibrating software reliability models", *IEEE Trans Software Engineering*, 16 (4), pp.458-70, 1990.

[Brocklehurst 1992] S. Brocklehurst, B. Littlewood, "New ways to get accurate reliability measures", *IEEE Software*, 9 (4), pp.34-42, 1992.

[Brocklehurst 1994] S. Brocklehurst, B. Littlewood, "Techniques for prediction analysis and recalibration", in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), McGraw-Hill and IEEE Computer Society Press, 1994.

[Chen 1996] S. Chen, S. Mills, "A Binary Markov Process Model for Random Testing", *IEEE Transactions on Software Engineering,* 22 (3) 1996.

[Cheung 1980] R. C. Cheung, "A User-Oriented Software Reliability Model", *IEEE Trans. on Software Engineering*, 6, pp.118-25, 1980.

[Chillarege 1994] R. Chillarege, "Orthogonal Defect Classification", in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.359-400, McGraw-Hill and IEEE Computer Society Press, 1994.

[Cochran 1977] W. G. Cochran, *Sampling techniques,* p., Wiley, New York, 1977.

[Cox 1979] D. R. Cox, D. Hinkley, *Theoretical Statistics,* p., Chapman and Hall, London, 1979.

[Crow 1977] L. H. Crow, *Confidence interval procedures for reliability growth analysis,* US Army Materiel Systems Analysis Activity, Aberdeen, Maryland, Tech Report, N°197, 1977.

[Currit 1986] P. A. Currit, M. Dyer, H. D. Mills, "Certifying the Reliability of Software", *IEEE TSE,* SE-12 (1), pp.3-11, 1986.

[Dawid 1984] A. P. Dawid, "Statistical theory: the prequential approach", *J Royal Statist Soc, A*, 147, pp.278-92, 1984.

[DeGroot 1986] M. H. DeGroot, *Probability and Statistics*, Series in Statistics, p., Addison-Wesley, Reading, Mass, 1986.

[Delic 1997] K. A. Delic, F. Mazzanti, L. Strigini, “Formalising Engineering Judgement on Software Dependability via Belief Networks”, in *DCCA-6, Sixth IFIP International Working Conference on Dependable Computing for Critical Applications, "Can We Rely on Computers?"*, (Garmisch-Partenkirchen, Germany), p.to appear, 1997.

[Donnelly 1994] M. Donnelly, B. Everett, J. Musa, G. Wilson, “Best Current Practice of SRE”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.219-53, McGraw-Hill and IEEE Computer Society Press, 1994.

[Duran 1984] J. Duran, S. Ntafos, “An evaluation of random testing”, *IEEE Trans. on Soft. Eng.*, 10, pp.438-44, 1984.

[ESA 1991a] ESA, *“PSS-01-21, Software Product Assurance Requirements for ESA Space Systems”*, European Space Agency, 1991a.

[ESA 1991b] ESA, *“PSS-05-0, ESA software engineering standards”*, European Space Agency, Feb, 1991b.

[ESA 1993] ESA, *“PSS-01-213, Guide to software reliability and safety assurance for ESA spoace systems”*, European Space Agency, August, 1993.

[Farr 1994] W. Farr, “Software Reliability Modeling Survey”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.71-116, McGraw-Hill and IEEE Computer Society Press, 1994.

[Frankl 1997] P. Frankl, D. Hamlet, B. Littlewood, L. Strigini, “Choosing a Testing Method to Deliver Reliability”, in *19th International Conference on Software Engineering (ICSE'97)*, p.to appear, 1997.

[Geiger 1979] W. Geiger, L. Gmeiner, H. Trauboth, U. Voges, “Program Testing Techniques for Nuclear Reactor Protection Systems”, *Computer, August 1979*, p16., 1979.

[Gilb 1974] T. Gilb, *“Parallel Programming”*, Datamation, 1974 (October), 1974.

[Goel 1979] A. L. Goel, K. Okumoto, “Time-Dependent Error-Detection Rate Model for Software and Other Performance Measures”, *IEEE Trans. on Reliability*, 28 (3), pp.206-11, 1979.

[Goyal 1992] A. Goyal, P. Shahabuddin, P. Heidelberg, V. F. Nicola, P. W. Glynn, “A Unified Framework for Simulating Markovian Models of Highly Dependable Systems”, *IEEE Transactions on Computers*, 41 (1), pp.36-51, 1992.

[Hamlet 1990] D. Hamlet, R. Taylor, “Partition testing does not inspire confidence”, *IEEE Transactions on Software Engineering*, 16, pp.1402-11, 1990.

[Horgan 1994] J. R. Horgan, A. P. Mathur, “Software Testing and Reliability”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.531-64, McGraw-Hill and IEEE Computer Society Press, 1994.

[Huang 1995] Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, “Software Rejuvenation: Analysis, Module and Applications”, in *Twenty Fifth International Symposium on Fault Tolerant Computing, FTCS-25*, (Pasadena , California, U.S.A.), IEEE Computer Society Press, 1995.

[Hunns 1991] D. M. Hunns, N. Wainwright, “Software-based protection for Sizewell B: the regulator's perspective”, *Nuclear Engineering International*, September, pp.38-40, 1991.

[Iyer 1994] R. K. Iyer, I. Lee, “Measurement Based Analysis of Software Reliability”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.303-54, McGraw-Hill and IEEE Computer Society Press, 1994.

[Jelinski 1972] Z. Jelinski, P. B. Moranda, “Software Reliability Research”, in *Statistical Computer Performance Evaluation* (W. Freiberger, Ed.), pp.465-84, Academic Press, New York, 1972.

[Kanoun 1993] K. Kanoun, M. Kaaniche, J.-C. Laprie, S. Metge, “SoRel: A Tool for Reliability Growth Analysis and Prediction from Statistical Failure Data”, in *23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.654-9, 1993.

[Kanoun 1994] K. Kanoun, J.-C. Laprie, “Trend Analysis”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.401-36, McGraw-Hill and IEEE Computer Society Press, 1994.

[Keiller 1983] P. A. Keiller, B. Littlewood, D. R. Miller, A. Sofer, “Comparison of Software Reliability Predictions”, in *Proc. 13th Int. Symp. on Fault-Tolerant Computing (FTCS-13),* (Milan), pp.128-34, 1983.

[Laprie 1994] J.-C. Laprie, K. Kanoun, “Software Reliability and System Reliability”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.27-70, McGraw-Hill and IEEE Computer Society Press, 1994.

[Laprie 1991] J. C. Laprie (Ed.), *Dependability: Basic Concepts and Associated Terminology,* Dependable Computing and Fault-Tolerant Systems, Springer-Verlag, 1991.

[Littlewood 1979] B. Littlewood, “Software reliability model for modular program structure”, *IEEE Trans Reliability,* 28 (3), pp.241-6, 1979.

[Littlewood 1981] B. Littlewood, “Stochastic Reliability Growth: A model for fault removal in computer programs and hardware designs”, *IEEE Trans. on Reliability*, 30, pp.313-20, 1981.

[Littlewood 1993] B. Littlewood, L. Strigini, “Assessment of ultra-high dependability for software-based systems”, *CACM,* 36 (11), pp.69-80, 1993.

[Littlewood 1973] B. Littlewood, J. L. Verrall, “A Bayesian Reliability Growth Model for Computer Software”, *J. Royal Statist. Soc. C,* 22, pp.332-46, 1973.

[Lyu 1995] M. R. Lyu (Ed.), *Software Fault Tolerance,* Trends in Software, Wiley, 1995.

[Lyu 1996] M. R. Lyu (Ed.), *Handbook of Software Reliability Engineering,* IEEE Computer Society Press and McGraw-Hill, 1996.

[Lyu 1992] M. R. Lyu, A. Nikora, “CASRE - A Computer-Aided Software Reliability Estimation Tool”, in *1992 Computer-Aided Software Engineering Workshop,* (Montreal, Canada), pp.264-75, 1992.

[Mann 1974] N. R. Mann, R. E. Schafer, N. D. Singpurwalla, *Methods for Statistical Analysis of Reliability and Life Data,* Wiley series in probability and mathematical statistics, p., John Wiley & Sons, New York, 1974.

[Martz 1982] H. F. Martz, R. A. Waller, *Bayesian Reliability Analysis,* Wiley series in probability and mathematical statistics, p., John Wiley & Sons, New York, 1982.

[May 1995a] J. May, G. Hughes, A. D. Lunn, “Reliability estimation from appropriate testing of plant protection software”, *Software Engineering Journal*, 10 (6), pp.206-18, 1995a.

[May 1995b] J. May, A. D. Lunn, “A model of code sharing for estimating software failure on demand probabilities”, *IEEE Trans. on Software Engineering Journal*, 21 (9), pp.747-53, 1995b.

[Miller 1986] D. R. Miller, A. Sofer, “A non-parametric approach to software reliability using complete monotonicity”, in *Software Reliability: State of the Art Report* (A. Bendell and P. Mellor, Eds.), 14:2, pp.183-95, Pergamon Infotech, London, 1986.

[Miller 1992] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, J. M. Voas, “Estimating the Probability of Failure When Testing Reveals No Failures”, *IEEE Transactions on Software Engineering*, 18 (1), pp.33-43, 1992.

[Musa 1979] J. Musa, *Software Reliability Data*, Rome Air Development Center, N.Y., Technical Report, 1979.

[Musa 1994] J. Musa, B. Juhlin, G. Fuoco, D. Kropfl, N. Irving, “The Operational Profile”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.167-216, McGraw-Hill and IEEE Computer Society Press, 1994.

[Musa 1993] J. D. Musa, “Operational Profiles in Software-Reliability Engineering”, *IEEE Software*, March, pp.14-32, 1993.

[Musa 1984] J. D. Musa, K. Okumoto, “A Logarithmic Poisson Execution Time Model for Software Reliability Measurement”, in *Proc. Compsac 84*, (Chicago), pp.230-8, 1984.

[Myers 1979] G. J. Myers, *The Art of Software Testing*, p., Wiley, New York, 1979.

[Nagel 1981] P. M. Nagel, J. A. Skrivan, *Software reliability: repetitive run experimentation and modelling*, Boeing Computer Services Company, Tech Report, N°BCS-40399, Dec 1981 1981.

[Nikora 1994] A. Nikora, M. R. Lyu, “Software Reliability Measurement Experience”, in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp.255-301, McGraw-Hill and IEEE Computer Society Press, 1994.

[NSWC 1993] NSWC, *Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) User's Guide*, Naval Surface Weapons Center, NSWC TR, N°84-373, Revision 3, 1993 (Dahlgren, Virginia, 22448-5000).

[Podgurski 1993] A. Podgurski, C. Yang, W. Masri, “Partition testing, stratified sampling, and cluster analysis”, in *ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, (Los Angeles), pp.169-81, ACM Press, 1993.

[Selby 1987] R. Selby, V. R. Basili, T. Baker, “Cleanroom Software Development: An Empirical Evaluation”, *IEEE Transactions on Software Engineering*, SE-13 (9), pp.1027-37, 1987.

[Siegrist 1988a] K. Siegrist, “Reliability of systems with Markov transfers of control”, *IEEE Trans Software Engineering*, 14 (7), pp.1049-53, 1988a.

[Siegrist 1988b] K. Siegrist, “Reliability of systems with Markov transfers of control, II”, *IEEE Trans Software Engineering*, 14 (10), pp.1478-80, 1988b.

[Smith 1993] A. F. M. Smith, G. O. Roberts, “Bayesian computation via the Gibbs sampler and related Markov chain Monte Carlo methods”, *J R Statist Soc B*, 55 (1), pp.3-23, 1993.

[Sova 1996] D. W. Sova, C. Smidts, “Increasing testing productivity and quality: a comparison of software testing methodologies within NASA”, *Empirical Software Engineering*, 1 (2), pp.165-88, 1996.

[Strigini 1996] L. Strigini, “On testing process control software for reliability assessment: the effects of correlation between successive failures”, *Software Testing Verification and Reliability*, 6 (1), pp.36-48, 1996.

[Titterington 1995] G. Titterington (Ed.), *Ovum Evaluates: Software Testing Tools,* Reports in Application Development, Ovum, 1995.

[Whittaker 1994] J. A. Whittaker, M. G. Thomason, “A Markov Chain Model for Statistical Software Testing" by”, *IEEE Transactions on Software Engineering*, 20 (10), pp.812-24, 1994.

3. Background

This section introduces the main advantages of statistical testing in achieving software reliability and in assessing it, and introduces the issue of setting reliability requirements for software.

3.1 Statistical testing for reliability prediction

Many claims that software is ‘sufficiently’ reliable are based upon indirect evidence, for example of the quality of the development process. Unfortunately, this kind of reasoning is not very trustworthy. Whilst good practices are probably necessary to obtain reliable software, their use cannot guarantee reliability: at best it will be possible to claim that they produce reliable software ‘on average’ or ‘usually’. In many situations, however, it is important to be sure that a *particular* program has achieved its reliability goal: in such cases we need a measure of the reliability that has actually been achieved.

The only means of measuring the reliability of software *directly* depend upon being able to observe, for a sufficiently long time, its behaviour when exercised in operation. Informally, if we observe a program executing failure-free for long enough (or if a sufficiently small number of failures are observed during a long period of operation), it would be reasonable to accept claims that it is sufficiently reliable for its intended purpose. More precisely, and subject to quite plausible assumptions, such data allow us to make accurate, *quantitative* assessments of reliability.

With some systems, particularly critical ones, real operation cannot be used to obtain this kind of data: we need to know that a system is sufficiently reliable *before* we are prepared to permit its operational use. It will therefore be necessary to simulate operational use in a testing environment: this is what we understand by the terms *operational testing*, or *statistical testing*, or *software reliability engineering (SRE) testing*. The key idea here is that the selection of test cases is carried out in such a way that the probabilities of selection are the same as those in real operation, so that any statistical estimates of reliability in the testing environment can be assumed also to estimate the operational reliability.

Clearly, the difficulty of doing this will vary considerable from one application to another. In certain industries there is already considerable experience of creating realistic simulations of physical system behaviour - ‘iron birds’ in the aircraft industry, for example - and it might be expected that in these cases it would also be possible to conduct statistical tests of software in condition that accurately reproduce operational conditions. Even in the case of critical software applications, these often concern the control of well-understood physical systems for which accurate simulation of behaviour is possible. This contrasts with many business systems, for example, where the nature of the use to which the product will be put can only be predicted imperfectly (because the existence of the product in operational use stimulates the user to creative operations that were not envisaged by its designers).

Sequences of statistically representative test cases can be constructed in many different ways. In some applications, there may be similar products in operation, and scripts of operational use can be recorded directly. In others, it will be necessary to construct the *operational profile* - the probability distribution over the set of all input points - indirectly.

During statistical testing it is important to record the data accurately. The data will be of two types: time, and counts of failure events. The time variable to be used will depend upon the application, and must be chosen with care. In process control applications, for example, the time will probably be actual plant operating time, and our reliability might be expressed as a rate of occurrence of failures, or a mean time to failure. In a safety system which only has to operate when (infrequently) called upon, such as many plant protection systems, time will be a discrete count of demands, and the reliability might be expressed as a probability of failure upon demand.

Statistical testing can be used in two main ways - for reliability growth estimation, or for estimating reliability from evidence of failure-free working - as we shall see in later sections. In both cases, it should be emphasised that the accuracy of the results will depend crucially upon the accuracy with which the simulation represents the statistical properties of the real operational environment.

3.2 Statistical testing for improving reliability

The discussion so far has concentrated upon testing which is intended to allow us to evaluate the reliability of a program, since this is the main theme of the present document. It should be said, though, that much of the software testing literature is concerned with the role of testing as a means of *achieving* reliability. Testing of this kind is, of course, generally of little or no use for our evaluative purposes.

In most software developments, testing will play both these roles. It will be used as a means of finding faults in software, so that these can be eliminated and thus the reliability increased; and it will be used to evaluate the reliability of the software. There has been considerable discussion in the scientific literature about the efficacy of statistical testing in the first of these roles. It has long been argued, by some, that statistical testing is a very inefficient means of finding software faults [Myers 1979]**.** If this is indeed the case, then (at least) two different kinds of testing will need to be carried out: one that is best at finding faults, and one (statistical testing) that allows reliability to be measured. In fact, there is evidence that statistical testing is also quite efficient in some cases at *finding* faults in software [Duran 1984, Hamlet 1990].

Statistical testing as a means for removing bugs is often seen with suspicion. It is argued that choice of test cases by expert human testers may be much better targeted to discover bugs than a pure random process. In fact, some testing procedures are precisely aimed at forcing software into unusual situations - for example, to test extremes of variables. However, it can be very misleading to judge the ability of a testing method to achieve reliability by its ability to find many faults for a particular expenditure of effort: we need to take account also of the *‘sizes’* of the faults found, i.e., the frequency with which they tend to cause failures. There is empirical evidence, from both experiments [Nagel 1981] and analysis of field data [Adams 1984], that these vary enormously: Adams, for example, found that some 30% of the faults found in the systems he studied would each show itself less than once every 5000 years of operational use. Clearly, any testing procedure that was efficient at finding these very 'small' faults, but inefficient at finding 'larger' ones (causing failures with higher frequencies), would not allow us to increase reliability efficiently.

In such cases, the fact that many faults are uncovered (and removed), for instance by boundary value testing, should not be taken to imply that there has necessarily been a large improvement in reliability - the faults may all be ‘small’ ones (i.e., they would seldom cause failures). Statistical testing, on the other hand, will expose faults with probabilities that are proportional to their size: if fault A has a greater contribution to the unreliability of the software than fault B, it will have a greater chance of being detected during a particular operational exposure. In fact, statistical testing is the only kind of testing which has this property of finding faults with probabilities that match their importance for operational reliability.

Of course, this observation does not mean that statistical testing is necessarily more efficient overall at improving reliability than other kinds of testing. It does, however, raise questions about claims made for the ‘obvious’ superiority of other types of testing in achieving reliability. That is, even if more directed forms of testing were more effective, for a specific kind of software and phase in the software life cycle, in discovering bugs, they might still be less effective in improving the software under test. As things stand, we need more empirical information on the relative merits of different testing methods in reliability improvement. An impressive record is that of the use of statistical testing at AT&T, as part of the practice dubbed "Software Reliability Engineering" in that company, reported to produce, for instance, a factor-of-10 reduction in customer-reported problems (more details will be given in Section 5.1). For a more thorough explanation of the relative effectiveness of statistical testing vs other testing methods in improving reliability, readers are referred to [Frankl 1997].

3.3 Deriving reliability targets for software components

Software reliability is not of interest in itself, but in that it contributes to determining whether a larger system, of which the software is part, will provide the required service. In the simplest case, the larger system must include the computer hardware which allows the software to run and to interact with its users (humans or machines). In the case of embedded computer systems, we may have both the case of software components being completely encapsulated in "black boxes" (e.g., an "intelligent" sensor or actuator) and the case of software interacting with hardware in complex ways, e.g., where software performs hardware diagnosis and reconfiguration, which in turn determines demands on the software.

Allocating reliability objectives (requirements) to software components should be just a part of the general process of allocating reliability goals to any other component[1] of a system. Its purpose is to state requirements for the developers of the modules to follow, distribute responsibilities within the organisation, and allow control of the development process. The requirements for each component are, in one form or another, upper bounds on the probability of undesired events (failures) originating in that component. Failures in any component may cause failures in the system. The system designer effectively has a budget of acceptable unreliability for the whole system, which must be allocated among the various components.

The process itself is conceptually the same as in any other reliability engineering process (section 6 gives more details including aspects that are specific to software). It starts with establishing reliability goals for the system of interest as a whole and building a rough design of the system, allowing one to determine how failures of components (including software components, if any) affect (cause failures in) the system. On these bases, a reliability model can be built, so that the reliability measures for the whole system are described as functions of the reliability measures for the components.

Reliability requirements are then allocated to the components, such that it can be shown (via the reliability model) that satisfying these requirements satisfies the reliability requirements for the whole system. In the general case, many such allocations are possible; sometimes, none is, and a re-design is necessary, and/or reduced targets for the whole system must be accepted.

Notice that the allocation must be *feasible*, i.e., the goals set for the components must be both *achievable* and *verifiable*. There must be reasonable confidence that the objectives can be reached and, for any system with any degree of criticality, that reasonable confidence that the objectives *have been* reached can be obtained before operation. It must be stressed that certainty can never be reached: every project carries a certain risk that the reliability target will not be achieved. There are methods for decreasing risk (increasing confidence), and they add to the cost of a project (statistical testing is one of them). What amounts to "reasonable confidence" thus varies among projects, and depends on the project risk that is considered acceptable (clearly, the risk that is accepted in developing, say, race cars is much higher than that acceptable in developing family cars). Reliability theory allows decision makers to choose their preferred trade-off between cost and uncertainty.

Methods for assigning reliability requirements are described in more detail in Section 6.2.

---

1 We are using the terms *component* and *subsystem* interchangeably, and *system* in the loosest sense: a system is made up of components, each of which can in turn be seen as a system made up of components. It has been observed that software is not, properly speaking, a system component, but merely the data stored in the program memory of computers in a system. This is true but does not invalidate the modelling of software as a component for the purpose of reasoning about system reliability: "a component" simply means "something that may cause events related to system failure". Likewise, the quality of a welding in a steel structure cannot be called a component of the structure, and yet it makes sense to discuss the probability that the quality is such that a crack may develop, or that a crack will actually develop in a certain time window. Similar arguments apply to other "non components", e.g. the operator's manual of a piece of equipment or the operating habits actually developed by operators.

4. Detailed conceptual description and terminology

In this section, we introduce a reference scenario and the meanings we are going to use for the essential terms in the rest of this document. Our terminology is largely taken from that developed by IFIP Working Group 10.4 [Laprie 1991]. The reader must be warned that most commonly used terms are used with different meanings by different parts of the technical community, and none of the many standard glossaries produced to date is likely to change this situation: hence the need to specify our use of the terms at the beginning of this document.

Glossary

Note: we have kept this glossary to a minimum, including: i) some words in general use that may have multiple meanings and we adopt with a specific restricted meaning to simplify technical explanations; ii) some technical words from the fields of reliability, probability, statistics.

Availability: a measure of how likely a system is to be available for operation at a given moment (point-wise, or instantaneous availability), or on average over a long time (steady-state, or asymptotic availability). It is affected both by the reliability of the system and by how promptly it is restored to service after failing.

Coverage:
test coverage (e.g., branch coverage, path coverage...): a measure of how much of the code of a program has been executed in a test campaign. Not relevant for this document; *oracle coverage*: the probability that a test oracle will detect what it is designed to detect (either software failure, or any erroneous value in the variables that the oracle can observe).

Dependability: a generic term encompassing multiple aspects of what makes a system trustworthy (reliability, availability, etc.). In this document, the word *reliability* is normally used in its place.

Error: variously used to designate *human error* (e.g., causing a bug in a program) or *erroneous state*. of the software.

Failure: the event in which a system stops delivering its intended service.

Fault: something that causes a system to fail, e.g. a software bug.

Faulty: containing (one or more) faults.

Independent events, independent random variables: Two events A and B are statistically independent if the following relationship holds among their probabilities: Pr(A and B)=Pr(A) Pr(B) or, equivalently, Pr(A|B)=Pr(A), or Pr(B|A)=Pr(B)

Input case, input point: a set of values, one for each of the variables that affect the behaviour of the software under test, and which the testers control, for a given test run (or the identical stretch of execution occurring outside testing), including single values or sequences of values for explicit input variables, state variables, state variables of the environment, time durations.

Input space: the set of all possible input cases

Input vector: a set of variables that it is convenient to represent as a vector. In particular, the set of values of inputs and state variables for one frame of execution of a repetitively run program unit (an input case may be a *sequence* (trajectory) of input vectors).

Operational profile: the set of the probabilities of all input cases for a given software product in a given spectrum of uses.

Oracle: any entity that decides whether a test run is a success or a failure.

Poisson process: a kind of stochastic process. In a continuously operating system, it is usually assumed that failure occurrences follow a Poisson, or, in a sense, are "purely random": the probability of failure in any very short interval, starting from a non-failed state, is equal to the duration of the interval times a constant "failure rate" λ.

Reliability: a) a generic term for the qualities that make a system trustworthy (also called dependability, integrity, assurance); b) the probability R(t) that a system does not fail within time t. In this document, this word is used with both meanings, depending on the context to avoid ambiguity.

Size (of a fault): the probability that the fault causes the software to fail on a test case chosen at random. It depends on the probability distribution of the input cases (operational profile).

Statistical testing: a form of software testing in which the test cases are produced by a random process meant to produce different test cases with the same probabilities with which they would arise in actual use of the software.

System: a generic term for a set of interacting entities. A system is made up of *components* (which we also call *subsystems*) which are systems in their own turn, i.e., they can be studied in terms of their component elements and their interaction. For instance, a spacecraft is a system; its attitude control system is a system; the software in the latter is a system; and the spacecraft, the rocket launching it, the launch facilities and mission control facilities together make up a system.

Test campaign: a series of test runs which it is useful to identify as such, e.g. the whole set of test runs performed on a software product during the acceptance process.

Test failure: the event in which software behaves in violation of its specification during a test run; the test run itself in which the violation occurs.

Test harness: the set of software and hardware tools supporting the execution of software for testing purposes, test case generation, recording of results, etc..

Test input: generic term for the information input to software during testing.

Test run: a period of execution of the software that is a useful unit of testing. It must be possible to decide whether a test run is a success or failure.

Test success: the event in which software behaves according to its specification for the duration of a test run; the test run itself.

Trajectory: a series of successive values for the input and state variables of a program that occurs in the operation of the software.

Typical abbreviations

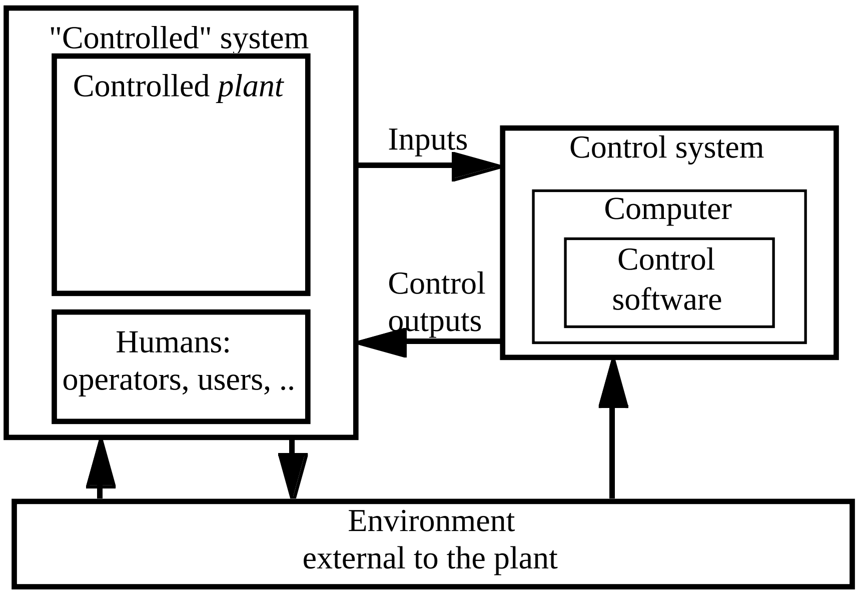
Cdf:	cumulative distribution function
E(X)	expected value (or *mean*) of the random variable X, defined as $\int_{-\infty}^{+\infty} x\ f(x)\ dx$
f(t)	probability density function of a random variable *T* , defined as $\frac{dF(t)}{dt}$
F(t)	[cumulative] distribution function of a random variable *T* , defined as Pr(T≤t)
Pdf:	probability density function
Pfd:	probability of failure per demand;
P(x), Pr(x):	probability of event x
r.v.:	random variable
R(t)	reliability at time t . If T is the r.v. "time to failure", *R(t) = Pr(T > t)* = 1 - *F(t)*
SUT:	software under test
t, T	a time variable (usually, time to failure, or number of test runs)
${}^{n}C_{r}$, or $\binom{n}{r}$:	binomial coefficient, $\frac{n!}{r!\ (n-r)!}$

λ	a value for a failure rate
ϑ	a value of a probability of failure per demand

4.1 Testing

4.1.1. Testing scenario, basic definitions

To discuss software reliability, it is necessary to consider the *use* of the software. The figure below depicts a typical situation:



Software operation scenario

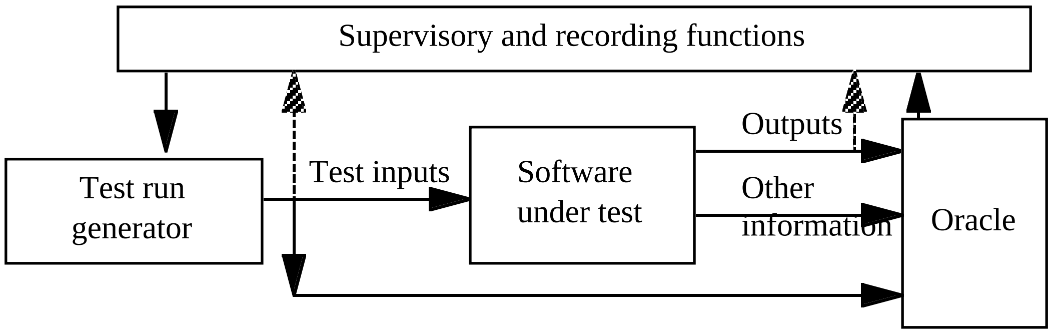
The software system we consider is part (together, possibly, with other software) of a *control* system[2], which may include digital computers as well as non-computer parts. The control system interacts with humans (operators, users, maintenance personnel, etc.: different roles exist depending on the system we consider) and a "controlled plant", which may be a spacecraft, an automobile, a chemical plant, etc. All these entities are part of a designed system (e.g., the system supporting a space mission), interacting with a pre-existing, "external" environment (which affects the designed system, e.g. via meteorological conditions, the state of the public power grid, objects in orbit, etc.).

All the items that are not part of the software form the *environment* in which the software operates, i.e., which affect its behaviour. This "environment" may include computer and non-computer hardware (or other software systems, either in the "control" or in the "controlled" system), humans, and, last but not least, the external environment.

[2] We shall use the same terminology for all kinds of software applications. For instance, the situation of software used in a safety system (e.g., an interlock system) will differ from the one shown here in the *purpose* of the software, and hence in its requirements (both in terms of required function and required dependability) but will still encompass both humans and engineered components acting under the combined influence of an external environment and of the safety system. A decision support system may act in an environment where the "controlled plant" is missing altogether; etc.

There is no specific rule for modelling a complex system in terms like those in the figure above. For instance, some organisations consider the sensors which convey plant information to a control computer to be part of the controlled plant, others that it is part of the control system outside the computer and others of the computer itself. What *is necessary* is that the same subdivision of the complete system into subsystems be used by all parties that must interact in achieving and evaluating reliability.

The requirements on a software system are dictated by its use in its intended environment of operation. Testing is one way of checking conformance to these requirements. Testing may be performed in the intended operational environment, but also in different environments.



Testing scenario

The figure above depicts a general scenario of the testing of a piece of software. The testers execute the *software under test* by feeding *test inputs* to it. The *test run generator* may be a human tester or a machine or software tool. A *test run* is a unit of execution such that it is meaningful to ask whether the software performed correctly during the test run. So, a test run will usually include the delivery of some input to the software under test, and the production of some output by it (we will see that for practical use in statistical testing, we will require test runs to be defined so that their characteristics are statistically *independent.*).

In practice, there are two typical ways of generating test runs:

- at each test run, the software under test and the test run generator are started anew and run until a natural unit of execution (e.g., an execution of a batch program, a mission or mission phase) terminates, or until the software fails.

- the software under test and test run generator are started and then left running for a long time. Test runs are defined as natural units of execution during this long execution sequence (e.g., responses to individual commands, transactions, frames of repetitive execution). In the limiting case in which the software is assumed to fail at constant failure rate in continuous time, a test run could be said to have infinitesimal length).

The word "input" is commonly used with many different meanings: to mean "input information" (as a non-countable noun); to mean "a single input variable", as defined in the source code; to mean "the event of software receiving information" (e.g., a message, an interrupt, a keystroke). When confusion may arise, we will use the term *input case* (for a test run) to mean all the input variables that are provided to the software for one test run. A value of the input case is thus a set of one value for each of the variables which form the input case. For instance, a piece of navigation software may receive, at each test run, new values for the co-ordinates of the position and the components of the speed of a spacecraft. An alternative term is *input point*: this emphasises the fact that the set of the variables that define an input case can be seen as the co-ordinates in a multi-dimensional "space" (the *input space*) in which each input case represents an individual "point".

An *oracle* (again, either human or automated) examines the output information (and possibly other information: the time the outputs are delivered, the values of some internal variables) produced by the software under test and decides whether this information is correct, in view of the input[s] to that test run. If this is the case (i.e., if the information is all and only that which the software is required to produce for that test run, and is output at the right time), we say that the test is a *success*. Otherwise, we say that there has been a *test failure*. The specification of an oracle is that it must correctly recognise successes from failures. In practice, oracles are usually not perfect: they may signal a failure when the software under test has performed correctly, and they may accept as a success a test run in which the software did not perform correctly.

A software *failure*, in test or in actual operation, is thus any discrepancy between the actual outputs and those that are required. If we look for what, in the software, caused the failure, we will usually find that the software is so built as to cause that failure[3] when subject to the same situation (input information). We will then say that the software is *faulty*, or that it contains one or more *faults*. As a result of the faults, certain executions of the software may develop differently from what the designer intended, e.g., they may assign to its variables different values from those intended. We then say that the state of the software is *erroneous* (or that it contains errors). "Error propagation" takes place within the software as more aspects of the software's execution become erroneous, e.g., when erroneous variable values are used as inputs to further operation of the software and cause erroneous values to be assigned to other variables. If, as a result, the outputs of the software violate its requirements, we say that the fault has caused a failure. From the point of view of any system encompassing the software in question, the failure is an internal error, which may propagate to other components and cause a system failure.

Note: Failures and errors may propagate among software components via unintended channels. For instance, if two components share the same computer platform, one may cause the other one to fail by overwriting some of its memory space, or by "stealing" CPU cycles from it, etc.

4.1.2. Range of input variables

The set of all the possible values for an input case will be called the *input space* of the software. For instance, if the software is a procedure with just two parameters, A and B, then the inputs space is the set of all the possible pairs of one value of A and one value of B. It can be described as a subset of the Cartesian plane. It is a subset, because any computer variable may only take a finite number of values. If there are more than two input variables, the input space will be a multi-dimensional "hyperspace".

It is usually the case that some possible values for the input case of a program are "illegal", i.e., the program is not meant to receive those values. For instance, for a function:

```
real arcsin(number)
```

computing the arcsin trigonometric function, all inputs outside the [-1,+1] interval are usually illegal; for a function

```
real power(base, exponent)
```

which computes the value of `base` raised to the power `exponent`, all pairs where `base` is negative and `exponent` is non-integer will be illegal. Whether these values should be considered as part of the input space is a practical decision of the testers, depending on whether they wish to test the behaviour of the software when used "illegally", or not.

3 Or to make that failure possible. Some failure behaviours only occur occasionally, and it may even be unfeasible to reproduce them in the laboratory, because they are only triggered by unusual states of the platform executing the software, like special load patterns, exhaustion of space for some operating system data structures. For the purpose of understanding software behaviour, it is appropriate to consider all these factors as part of the input case submitted to the software. For the purpose of generating test runs, we are often compelled to depend on mechanisms that we cannot directly control (e.g., the behaviour of the operating system under overload conditions) to reproduce these factors.

4.1.3. What is an input case?

It is important to consider that an input case may take many forms, depending on the type of software that we consider:

- the simplest case is that of a "batch" program, which receives all of its input information at the beginning of its execution, runs for a while and then produces all of its output information. For instance, a procedure call receives values for all its arguments at the moment the procedure is activated. So, we can easily specify the input vector for a given test run by saying, for instance, that `argument_1` has the value "23", `argument_2` has the value "12", and so on. We could then say that the input variables form an *input vector*, and a test run consists of invoking the software under test with a certain input vector. The set of all possible values of an input vector form the *input space* for the software;

- even for a batch program, there are factors that affect the behaviour of the software although they are not written in its internal variables. For instance, the state of the process queue in a computer may decide how quickly the software under test is executed, and thus whether it completes a test run on time or too late (and thus fails). Other factors are memory contents left over by previous execution of other programs, possible hardware faults (e.g., a faulty memory cell used for storing a program variable), and so on. When we wish to test for the effects of these factors, we must consider them as input variables to the program;

- in other kinds of programs (especially control programs), the values for the variables (i.e., the components of the input case) may be input at different times, as for instance in a vending machine which receives first the buyer's choice of goods and then the information about the coins inserted. So, we need to specify an input case as a sequence "first, `variable_1` receives the value '23', then `variable_2` receives the value '7', then ....";

- the components of the input case may even be multiple values for the same program variable, as in a control program whose internal variables are updated at each frame of execution with new values. In this case, we may have to specify an input case in a more complex way, for instance "`argument_1` has the value 23 followed by 12 followed by ...". If we represent the input vector for one frame of execution as one point in a multidimensional space, a whole input case will be described by a *trajectory* in this space. If there are, for instance, two input variables, and an input case consists of, say, 100 frames of execution, at each one of which the two variables are read again, the input space is a 200-dimension space (in mathematical terms, it is said to be the *Cartesian product* of the 2-dimensional space by itself, for 100 times);

- even more complex situations are possible. For instance, for an Ada task exporting several entry points we may have to define an input case as the sequence of intervals between successive calls, plus the name of the entry point called each time, plus the values of the parameters of the call. The same description applies to a HOOD object with the various operations it exports.

Many programs have an internal state which they change as a consequence of receiving inputs. The program's outputs are a function of its internal state at a given moment, and of the input information it receives from that moment and until it produces the outputs. There are two ways for representing this:

- we may define a test run as consisting of initialising the state variables of the program to their specified initial values (as specified in the program), and then providing a sequence of values for input variables. In this case, the "input case" is naturally defined as this sequence of values, as in the examples in the previous paragraph;

- on the other hand, we may organise a test run by first setting the internal variables (the state) of the software under test to specific values (different from their normal initialisation values), to simulate the effects of a previous period of execution, and *then* submitting a sequence of input values. In this case, the input case consists of the values imposed to the state variables, plus the sequence of input values.

4.2 A conceptual model of the software failure process

4.2.1. Hardware and software failure processes: an inherent uncertainty

It is a truism that interest in reliability always concerns *systems*, and these will usually contain both hardware and software. Nevertheless, it has become common practice to talk of "hardware reliability" and "software reliability". It is useful to begin with a brief discussion of the differences and similarities between reliability concepts for software-based systems and for purely hardware systems, not least because the scientific and engineering literature on reliability is traditionally associated with hardware only.

A distinction is sometimes made between *random failure* (e.g. due to hardware component failure) and *systematic failure* (e.g. due to design faults being uncovered in hardware or software). This distinction is somewhat misleading, inasmuch as the use of the words "random" and "systematic" seems to suggest that in the one case a probabilistic approach is inevitable, but that in the other we might be able to use completely deterministic reasoning. In fact this is not the case, and probabilistic arguments seem inevitable in both cases.

When we use the word *systematic* here it refers to the fault mechanism, i.e. the mechanism whereby a fault reveals itself as a failure, and not to the failure *process*. Thus it is correct to say that if a fault of this class has shown itself in certain circumstances, then it can be guaranteed to show itself whenever these circumstances are exactly reproduced. In the terminology of software, which is usually the most important source of systematic failures, we would say that if a program failed once on a particular input case it would always fail on that input case until the offending fault had been successfully removed. In this sense there is determinism, and it is from this determinism that we obtain the terminology.

However, our interest really centres upon the failure *process*: what we see when the system under study is in operational use. In a real-time system, for example, we would have a well-defined time variable (not necessarily real clock time) and our interest would centre upon the process of failures as time progresses. We might wish to assure ourselves that the rate of occurrence of failures was sufficiently small, or that there was a sufficiently high probability of surviving some pre-assigned mission time. The important point is that this failure process is not deterministic for "systematic" faults, since there is an inevitable uncertainty associated with the selection of the inputs.

We shall use the terminology of software here, but it should be remembered that systematic failures also include those arising from certain design faults in hardware. Indeed, the very success of conventional hardware reliability engineering against faults that arise from physical decay is now revealing the importance of design faults to the overall reliability of complex systems, even those which do not contain software. Recent successes in devising intelligent strategies to minimise the effects of physical failure of components results in a higher proportion of even "hardware" failures being caused by flawed designs. Software, on the other hand, has *no* significant physical manifestation: its failures are always the result of inherent design faults revealing themselves under appropriate operational circumstances. These faults will have been resident in the software since their creation in the original design or in subsequent changes.

4.2.2. The input space, software faults and failures

The software failure process, as we have seen, is the process of revealing of faults as a result of executions on input cases. We call the totality of all possible input cases that might ever be received by the software "the *input space"*, and this is often extremely large and even, sometimes, ill-defined. The precise nature of an "input" will vary from one application to another, as shown previously. The important point here is that "input" should be sufficiently general to capture all information that is relevant, in the external environment and the internal state of the computer: it should be defined in such a way that the same input will always produce the same output (so long as the program does not change). In practice, programs often appear to behave non-deterministically, as explained before, in that their reaction to a given input case depends on variables that have not been observed, and are possibly very difficult to observe. When testing, we shall often have to drive the production of input cases by specifying values for those variables that are under our direct control, and depending on less controllable processes (e.g., tuning the load of the target computer) to (statistically) control the other variables.

For a particular program, each input case will fall into one of two mutually exclusive classes: those that, when executed, always produce acceptable output, and those that do not. When the software produces unacceptable outputs, we say that one of the latter input cases (a *failure point*) was executed. Clearly, there must be a means of deciding whether output is acceptable or not: for each input it must be possible to decide whether the output is in accord with what the specification required. It is worth noting that this is often an area where there is some degree of judgement needed, but we shall assume that in all cases such an *oracle* exists, and is consistent in always giving the same answer when presented with the same information.

Every time a successful fix is performed on the software, a set of failure points (a *failure region*) is eliminated. The probability of failure of the software is thus reduced by an amount equal to the probability that any of those points is executed. So, we can say that the fix removed a fault. The *size* of that fault, i.e., the probability of selecting a point from the corresponding failure region, is the amount by which the probability of failure of the software decreases as a result of the fix. Again, it is worth noticing that the effectiveness of testing for improving reliability is measured by the net total size of the faults removed (net, i.e., after detracting the sizes of faults added), not by the number of such faults, which should be more properly called the number of fixes made[4].

4.2.3. Operational profile

Software execution, then, can be seen as the selection of inputs from the input space and their transformation into outputs. As we have seen, there is usually inherent uncertainty about this selection mechanism arising from the nature of the operational environment. Thus, for example, in the case of a process control system there would be inherent uncertainty about which values the sensor readings will take, arising from elements of indeterminacy in the process being controlled. The particular *operational profile* will determine the probabilities of selection of the different inputs. Indeed, it is often the case that this profile will vary from one application of the software to another. Operating systems, for example, are known to have dramatically different profiles from one user to another (and hence very different user-perceived reliabilities); systems that interact with humans will often have reliabilities that depend upon the detailed nature of this human use.

The operational profile can be thought of as a probability distribution over the input space, i.e., an assignment of a probability to each point (input case) in the space. For a one-shot system, for example, the different possible demands will be represented by the points of the input space, and the operational profile will be completely specified if we know the probability of each demand at all

4 Notice that one tester might require a series of fixes, followed by new failures, to remove a failure region that another tester would remove by one, more enlightened fix after just one failure. Though the former tester "finds more faults", the latter is clearly the better one of the two. To avoid confusion, it is important to consider that the very concept of *a fault* is one that is only meaningful *after* the fact of fixing it. There are cases (e.g., typos) in which "the fault" that caused a failure seems self-evident, as a single discrepancy between the developers' intentions and their code. But in general, and especially for the more subtle failures of mature software, given a certain observed failure, different people will offer different fixes: fixes that affect different parts of the code, create different code, and affect different sets of failure points. So, we cannot even properly say that a program contains a given number of faults, but only that it is faulty.

times. This might be a reasonable approximation in the case of a reactor protection system, where we could assume that successive demands are a long time apart and thus are approximately statistically independent with the same probability distribution. For a continuously operating system, such as a real-time control system, it is often better to think of the operational profile as a stochastic process[5]. Thus inputs for a flight control system can be regarded as a vector of sensor readings that are cyclically presented to the computer software, which returns instructions to actuators. Here there would be strong correlations between successive vectors of readings, and we could think of the execution process as a trajectory through the input space. As we shall see, we will often require that this whole trajectory, or a long stretch of it, be regarded as a single input case.

4.2.4. The software failure process

As well as the uncertainty about the sequence of successive inputs, there is also uncertainty about the faults in the system, i.e., more precisely, of where the failure points (if any) are located in the input space. Clearly this is an additional source of uncertainty: we would not know which inputs, of the ones we had not yet executed, would produce a failure if executed, nor would we have any information about how these might cluster into "faults" as we have defined them.

There is uncertainty in the software failure process, then, for two main reasons: we do not know where the faults lie in the input space, and we do not know which inputs will be selected in the future. This uncertainty can only be captured by probabilistic representations of the failure process: *the use of probability-based measures to express our confidence in the reliability of a software-based system is therefore inevitable.*

So far we have only considered the very simplest case of a system not subject to "repair". If instead we consider the process of software failures in which fault-removal (debugging) is being attempted as the faults are revealed, further uncertainty arises. We cannot be certain that the fault truly has been removed; it may be the case that the fixing attempt has introduced novel sources of failure. Even in the event that the fault removal has been successful, we do not know by how much the reliability of the program has improved - i.e. we do not know the "sizes" of the newly discovered faults (there is, however, plenty empirical evidence that the contributions of different faults to system unreliability can vary by several orders of magnitude, see e.g. [Nagel 1981, Adams 1984]).

4.2.5. Probabilistic measures of reliability

Because of the inevitable uncertainty associated with the software failure process, we shall need to express our reliability requirements for a system (or a subsystem) probabilistically. The precise way we choose to express the reliability, as well as the numerical level, will to a large extent depend upon the nature of the application. Thus for a transaction processing system, such as an airline reservation system, the total amount of time it is down within a particular period would be more important than the *number of times* it was down. For a flight-critical system, on the other hand, it might be assumed that each failure was of potentially catastrophic consequence, so the frequency of these would be of most importance. In such a case it would be merely academic whether or not correct functioning could be quickly restored following a failure. In many cases, there is no real sense in which we can sensibly talk about *the* reliability of the system; rather we need to know of more than one measure. For an air traffic control system, for example, it would be important to know that failures which resulted in incorrect information being produced (and thus possible incorrect action) occurred sufficiently infrequently *and* that the system was available to serve new demands for a high proportion of the time that demands were being made upon it.

---

5 A *stochastic process* is given by a random variable (or a set of random variables) whose probability distribution is a function of time, and may depend on the history of the process prior to that time. At a set time *t*, each possible combination of values for the input variables has a certain probability, and these probabilities vary over time. A realisation of the stochastic process is a deterministic function of time: the sequences of values of the input variables observed during one specific execution.

In the following we give some examples of common ways in which the uncertainty about the correct functioning of a system might be expressed. We shall distinguish between continuous and discrete time-scales, although, as we shall see, this distinction can be somewhat arbitrary. For readers familiar with hardware reliability, these measures will not be new: although the *reasons* for unreliability in software are different from those in hardware systems, the basic unpredictability of the failure process ensures that the measures we use are compatible, thus enabling us to compute measures of *system* reliability

4.2.5.1. Instantaneous reliability

Here we are concerned with the answer to questions such as "how reliable is the system now?" i.e. at a specified moment in time. For systems that are expected to give continuous service in time, the simplest variable about which we might be interested is the *time to next failure*, T[6]. Clearly, this is a random variable, and we have a complete description of our uncertainty about it if we know its *distribution function, F(t),*

$$F(t) = Pr(T \leq t)$$

or its *reliability function*, *R(t)*:

$$R(t) = Pr(T > t) = 1 - F(t)$$

For a given value of *t*, the reliability function tells us the probability that the system will be able to work for that length of time without failing. If the application is one where there is a natural notion of "mission", as for example in the case of spacecraft, then *t* would be chosen to be the length of mission and we would obtain the probability that the spacecraft would survive a mission without failing.

In traditional hardware reliability studies it has been common to use a single attribute of this distribution, the *mean time to failure* or *MTTF*:

$$MTTF=E(T) = \int_0^\infty R(t)dt$$

However, this may not be particularly useful for cases where design faults are the source of failures. In the hardware case we often deal with an indefinite number of similar components, so that the MTTF can be thought of as the average life time of a large sample of components In the case of a system suffering from the effects of software design faults, however, it may well be the case that the system will be allowed to suffer from a particular fault only once, and then the fault will be fixed. In this case, there is thus no natural intuitive interpretation for the averaging in the above definition (which would exists if many copies of the software were run without fixing the faults). This does not detract from the mathematical meaningfulness of the MTTF, however.

4.2.5.2. Measures based on the failure process

One feature of design failures that distinguishes them from "random" failures is that repair is not usually necessary for the system to be set working again following a failure (although we may, of course, *choose* to carry out repair). Thus when a computer program fails on a particular input, there is no reason to believe that there is any increased chance that it will fail when presented with a different, unrelated input[7]. This contrasts with the common situation of a purely hardware system

[6] We adopt the convention of using upper case letters to represent *random variables*, and lower case to represent their realisations on specific trials.

[7] An important exception to this concerns the *clustering* of failures in many control applications, as successive inputs are *not* unrelated. A separate consideration is that, after a failure, it may be necessary to *recover* the erroneous internal state of the software; but this does not require one to correct the fault itself, i.e., the defect in the code.

which has failed because of the mechanical breakage of a component; here it is often the case that the system *cannot* work again until the failed component is replaced. It is for this reason that so much attention has been devoted in classical hardware reliability theory to the study of the life-time characteristics of systems, i.e. to the statistical properties of merely the time to (first) failure.

For software (and in general for "systematic" failures) we are more likely to be interested in a stochastic *process* of failure events in time. At its simplest, this would arise because the software was restarted at a new point in the input space and would then execute failure-free until the next time a failure region (perhaps the same one) was encountered. Of even more interest is the case where there are attempts to repair the system, since in the case of "systematic" failures this will (if successful) mean the removal of faults and therefore a growth in the reliability of the system. Once again this contrasts with the common case of random hardware failures, where the repair action merely restores the system to its earlier reliability level[8].

We thus see a sequence of failures, corresponding to different design faults being activated by the exercise of appropriate parts of the input space. In its simplest form (ignoring, for example, the times needed for the fixes of the design faults) this will be a *stochastic point process*: a random process of point events in time. Because of the attempts, at these failure events, to fix the faults, this process will be non-stationary (i.e., the probabilities of the various events will vary over time). It is in modelling this *reliability growth* scenario that most of the research in software reliability has concentrated in the past few years. The position now is that this problem is well-understood, with solutions that work well, at least for fairly modest levels of reliability.

These failure processes can be quite complex, and we now have a rich set of different reliability measures to characterise matters of practical interest. The *rate of occurrence of failures* (*ROCOF*), $r(t)$, is the instantaneous rate at which failures are occurring when a total time t has elapsed from the time origin. In mathematical terms, $r(t)dt$ is the probability of a failure occurring in the time interval $(t, t+dt]$ when dt is small.

At any particular moment during the observation of this process, we could be interested in the time to next failure, so the measures based upon the reliability function are also appropriate. However, care should be taken here to distinguish between global and local time: that is, between the time variable that represents total elapsed time from the beginning of observation of the process, and time measured from the present moment until the next event. In fact *all* the measures defined in the previous section will change with global time for these non-stationary processes.

A random variable of interest in this stochastic point process is the count, $N(t)$, of the accumulated number of failures in $(0, t)$. It is often the case that the distribution of this is very complex, but its mean

$$M(t) = E(N(t))$$

will sometimes suffice. The ROCOF, $r(t)$, is in fact the rate of change of $M(t)$ with respect to time: $r(t) = M'(t)$.

4.2.5.3. Discrete time reliability measures

For some applications, calendar time is an inappropriate variable upon which to base the reliability measures. Examples include safety systems that are only called upon to respond when some other system has failed, and certain transaction processing systems. In these cases, it is the number of demands that have been placed upon the system that is important, rather than the amount of clock time that has passed. Essentially, we seek a time base such that the "failure rate" of the software with respect to it can be considered approximately constant. Thus, for instance, calendar time is usually a less appropriate time base than CPU execution time because programs may be run only at

8 Repairable systems, with growing reliability, *have* been considered in the hardware context [Ascher, 1984 #591], but far more effort has gone into the study of life-time characteristics of non-repairable systems.

irregular intervals, and they can only fail while running; programs can be ported between platforms that are essentially identical, apart from different CPU speeds; and so on.

There are usually discrete time equivalents of the different reliability measures that are used in the continuous case. Thus the probability of failure upon demand can be seen as a discrete version of the ROCOF, where the number of successive demands to date forms the discrete time variable. The probability of surviving a specified number of demands without failure is analogous to the reliability function, and so on.

4.2.5.4. Availability

All the preceding measures are concerned with reliability, and essentially are determined by the frequency with which failures of the system are occurring. That is, they are concerned solely with *events* regarded as points in time. Availability, on the other hand, is concerned with time *intervals* of system loss following failure. Availability is determined by the combination of how often the system fails (i.e., by the system's reliability) and how quickly it is restored to service. "Restoration to service" implies that physical faults are repaired (which does not apply for failures caused by design faults, like software failures), and the erroneous internal state is corrected.

Once again, there are different ways in which our interest in the system being available can be captured, all of which are, confusingly, called availability. Thus point-wise availability is the probability that the system is available at time *t*, interval availability is the expected proportion of time the system will be available in the interval (0, *t*). Although these can differ numerically when we observe the system for only a short time, there are asymptotic results which cause them to take similar values over long periods.

4.2.5.5. Safety

Some of the possible failures of a system have particularly serious consequences (e.g. loss of life or limb, or of the system, or other kinds of damage outside the system). Freedom from such failures is usually called *safety*. Probabilistic requirements on safety can thus be described as reliability requirements concerning specific classes of failures (characterised as, "unsafe", "[potentially] dangerous", "catastrophic", ... failures).

When safety considerations are applied to software, which has no physical existence, it must be considered that all dangerous consequences take place through the action of software on its environment. This may make it difficult to specify in advance which failure modes have to be considered as "unsafe", especially if a software component is intended for use in various, different environments (other software components and/or controlled plants). This uncertainty may impose pessimistic assumptions. For instance, when a software component can cause unsafe behaviour in the controlled plant, it is often necessary to assume that all failures of the software component will cause such behaviour. Notice that such software may not be designed to have any direct authority to cause unsafe behaviour: its criticality may be simply due to its sharing a computer with software that does have such authority.

It is often advocated that safety concerns in a project be completely separated from reliability concerns. As far as statistical testing is concerned, however, the difference between safety and reliability is merely a difference between categories of failures, and most guidelines are the same irrespective of the class of failure concerned. In the rest of this document, we will only mention safety as separate from reliability where safety-relevant failures require different considerations from other failures.

4.2.5.6. Choice of time variable

Last, we must consider the choice of the time variable for reliability measures. We have already mentioned that it is often convenient to use "discrete time", typically a number of demands. This may apply, for instance, to some safety systems, transaction processing systems, and many forms of "batch" software, as well as to units of software that are evaluated without the rest of the software system they belong to. "Convenience", here, refers not only to the most natural way of specifying requirements and/or of organising testing, but also to what appears to be the "real" probabilistic behaviour of the software. For instance, if it is observed that the duration of execution of a procedure is not correlated with its observed failures, it appears natural to see the failure process as the results of many trials (demands), each with a set probability of failure, rather than as an effect of physical time. The physical process that causes failures appears more similar to a sequence of dice tosses than to the depletion of a water reservoir over time.

Similar considerations apply in the case of continuous time as well. So, wall-clock (or calendar) time is not always the best choice. One such consideration is that a specific software component is not going to fail while it is not in execution. Software components that share the same computer (e.g., procedures in the same program, or user programs and operating system programs in a time-shared system) are likely contribute to the failure probability of the system that they form in shares that are roughly proportional to their shares of the total CPU cycles and to their own "failure rates" *measured over execution time* rather than calendar time.

4.2.5.7. Discussion

The different categories of reliability measures above should not be seen as definitive, rather they are guidelines to common practice. It is always necessary to exercise judgement to choose the appropriate reliability measure(s) for a particular application. Thus, for example, a system may not fit neatly into the discrete-time/continuous-time dichotomy above. Some nuclear reactor control systems also have a protection function: in its control function, a continuous-time reliability goal would be appropriate, but for protection a discrete-time measure, such as probability of failure on demand, would be needed in addition. We have already mentioned the case of an Air Traffic Control system for which both reliability and availability would be relevant.

The subsection about safety, above, introduced the consideration that we often require separate reliability goals for different *types* of failure event. For example, in the case of a telephone switch, we might accept a fairly modest reliability goal for the loss of individual calls, but would require a high reliability against the loss of the complete switch (not least because this would inhibit emergency calls, and thus have safety implications). In the case of the reactor control system the protection function is clearly safety-critical, whereas the control function may not be.

These examples introduce a further notion of failure *severity* that is lost in a simplistic reliability analysis, which at best treats only the point process of failure events. This can be handled in several ways. The simplest approach is just, as above, to have different reliability levels for each of a set of categories of events. A more refined approach would be to assign a cost (or consequence) variable to each event, representing its severity, and then examine the probabilistic properties of the cost process in time. Thus we might ask, for example, what is the probability that the cost of failures in time t did not exceed c. In some cases, the cost might be described in terms of a money loss; other such variables may be used, and one could even use multiple, incommensurable cost functions (in some contexts, for instance, loss of life and money losses would not be converted into a common unit of measurement, and separate requirements would apply to the two forms of cost).

In this document, whenever we do not specify the kind of reliability measure (and time base) of interest, we will refer to the case in which the measure of interest is the probability of failure per run.

5. Experience of statistical testing - benefits, costs, limitations

5.1 Industrial experience

Software Engineering does not in general have a good record of empirical support for its practices and procedures - at least in comparison with other engineering disciplines. This stems partly from the rapid changes that continue to take place in methods and techniques, and partly from its comparatively short history - decades rather than centuries. Software Reliability Engineering (SRE), however, is one of the better-placed branches of software engineering, and there is now growing evidence of the efficacy of the new techniques, both from academic experiments and industrial experience.

Statistical testing is one of the elements of the "Cleanroom" approach [Currit 1986]. In the Cleanroom approach, developed to shift the emphasis in software development from debugging to avoiding the creation of bugs, statistical, independent testing is the only dynamic testing applied. Very strong claims have been made for the advantages of the Cleanroom approach in achieving reliability even *before* testing: it is difficult to separate out the experience about statistical testing [Selby 1987]. More recently, the Software Engineering Laboratory (SEL) at the NASA Goddard Space Flight Center has performed an experimental comparison [Sova 1996] of the results with three testing "methodologies" it uses, i.e., a "traditional" approach (with multiple phases of testing tied to the phases of a waterfall life cycle), a Cleanroom approach and a "modified" approach. This is a hybrid of the preceding two in that it massively reduces testing by the developers, compared to the traditional approach. The Cleanroom and "modified" approaches use statistical testing for acceptance, while the "traditional" approach uses a checklist of functional test items. The comparison was limited to two projects per method. Although comparing testing productivity between projects is difficult, this study reports that the Cleanroom and "modified" methods approximately halved the effort required to test the software. Defect density as "measured" at acceptance test was similar between the Cleanroom and traditional methods, and better with the "modified" method. In addition, the use of "operational scenario testing" was "felt" to be much more effective than testing by developers.

Good examples of the use of statistical testing and reliability evaluation come from the work of John Musa at AT&T Bell Labs, and from Michael Lyu at Jet Propulsion Laboratory and later at AT&T Bell Labs. Both report successful application of SRE on real industrial projects.

Musa reports ([Lyu 1996], Chapter 6) on his experience with a telephone system, AT&T's International DEFINITY, in which statistical testing was used in conjunction with other quality techniques. He reports that the quality improvement from the previous major release was dramatic:

- a factor-of-10 reduction in customer-reported problems
- a factor-of-10 reduction in program maintenance costs
- a factor-of-2 reduction in the system test interval
- a 30% reduction in new product introduction interval.

Musa claims important commercial advantages stemmed from the use of the overall approach, but he has particularly strong claims for the statistical testing. He reports that 20% of the operations represented 90% of the use of the system, 20% of the faults caused 95% of the failures, and that testing the 20% of high-usage operations first significantly speeded up the reliability improvement of the system.

Musa acknowledges that the most significant cost in applying SRE to this system development was in determining the operational profile in order to conduct statistical testing. Reporting on this and other projects in which he has been involved, he estimates that effort ranges from one person-week to one person-year, depending upon the size of the project. His experience is that the average effort for a project with 10 developers and 100,000 lines over 18 months is about 2 person-months. The very largest project, involving 1000 developers and multiple large operational profiles, required the effort of one person full time. Costs involved in analysing the data obtained from statistical testing are modest compared with the costs for creating the profile.

Lyu ([Lyu 1996], Chapter 7) reports on 5 space applications at JPL, and on a large (1 million lines of code) telecommunications application at Bellcore. His results confirm the evidence from others (e.g. [Brocklehurst 1992, Abdel-Ghaly 1986]) that accurate reliability predictions can be obtained in most circumstances: the most important point is that the operational environment should be captured as accurately as possible. Musa, however, points out that even rough approximations to this profile can be useful, and give superior results, in terms of reliability improvements, compared to conventional testing. As for reliability estimation, errors in the operational profile can in theory cause the estimation to be radically wrong (taking any value between 0 and 1). Some authors claim that the errors that are likely in practice will usually only produce small errors in estimation, but there is insufficient experience for a tester to know in advance all the approximations that can be allowed without causing excessive error.

A recent application of statistical testing was performed on the software for the Primary Protection System (PPS) of the Sizewell B nuclear power station [May 1995a]. In this case, statistical testing was required as an afterthought. It was run in addition to the licensing process for the reactor, but with support by the licensing authority. Although the process of dynamic testing (required for licensing) was expensive, in terms of developing a test harness with a satisfactory simulator for the reactor, the component of statistical management of the process did not significantly add to the cost.

5.2 Practical limitations: the feasibility of assessing ultra-high reliability

In section 9 we shall see in more detail how the failure data obtained from statistical testing can be used to obtain estimates and predictions of software reliability. In the meantime it is instructive to consider the levels of reliability that it is feasible to measure in this way, so as to gain an insight into the limits of these techniques. We begin with an example of some software failure data that have been used extensively in the literature. It is called S1, and was carefully collected during in-house statistical testing by John Musa at Bell Labs [Musa 1979].

3,	30,	113,	81,	115,	9,	2,	91,	112,	15,	138,	50,	77,	24,	108,	88,
670,	120,	26,	114,	325,	55,	242,	68,	422,	180,	10,	1146,	600,	15,	36,	4,
0,	8,	227,	65,	176,	58,	457,	300,	97,	263,	452,	255,	197,	193,	6,	79,
816,	1351,	148,	21,	233,	134,	357,	193,	236,	31,	369,	748,	0,	232,	330,	365,
1222,	543,	10,	16,	529,	379,	44,	129,	810,	290,	300,	529,	281,	160,	828,	1011,
445,	296,	1755,	1064,	1783,	860,	983,	707,	33,	868,	724,	2323,	2930,	1461,	843,	12,
261,	1800,	865,	1435,	30,	143,	108,	0,	3110,	1247,	943,	700,	875,	245,	729,	1897,
447,	386,	446,	122,	990,	948,	1082,	22,	75,	482,	5509,	100,	10,	1071,	371,	790,
6150,	3321,	1045,	648,	5485,	1160,	1864,	4116								

The table above gives the raw data, representing CPU *execution times* (in seconds) between successive failures. When a failure occurs during this testing, the underlying fault is identified and fixed before the test is restarted. The resulting reliability growth is obvious from a cursory glance at the data: there is a clear tendency for the later times to be larger than the earlier ones. The statistical techniques that will be described in section 9 allow this reliability growth to be tracked - that is, they enable the tester to estimate the current reliability at any time during the test, and to predict how this will change during later stages of test.

sample size, i	elapsed time, t_i	achieved MTTF, m_i	t_i/m_i
40	6380	288.8	22.1
50	10089	375.0	26.9
60	12560	392.5	32.0
70	16186	437.5	37.0
80	20567	490.4	41.9
90	29361	617.3	47.7
100	42015	776.3	54.1
110	49416	841.6	58.7
120	56485	896.4	63.0
130	74364	1054.1	70.1

The table above shows the kind of results that can be expected from this kind of analysis. Here the current reliability, here expressed as the mean time to failure, is estimated at different times during the debugging of the program in statistical test. The succession of estimates clearly shows reliability growth, as we would expect. However, what is notable is that the growth is quite slow. The second column in the table shows the total time of statistical testing required to achieve the *MTTF* given in the corresponding entry of the third column. The fourth column gives the ratio of these entries, and can be thought of as an estimate of a simple cost/benefit ratio. There are two things notable about the entries in this column. In the first place, the entries are quite large: if we have a particular target *MTTF* in mind, expressed as a certain number of seconds in this case, then we shall have to wait a large multiple of this number of seconds before we can be confident that the target has been reached. Secondly, this ratio is increasing, so that there is a strong "law of diminishing returns" for statistical testing: if our goal is a very large *MTTF*, we might have to test for an infeasibly long time.

Results like these will be observed on all data sets exhibiting reliability growth during operational testing, whatever the particular measure of reliability used. Clearly, there are quite stringent practical limitations to the levels of reliability that can be demonstrated in this way.

In the case of the most critical systems, whose failure could have catastrophic consequences, it may be necessary to have an even more stringent regime for calculating reliability estimates, with resulting implications for the reliability targets that can be regarded as feasible. The difficulty in using the reliability growth approach in these cases is that it assumes that faults are in fact removed when they are discovered (at least on average). In some safety-critical applications, e.g. nuclear [Hunns 1991], the more conservative approach is taken whereby *any* change to a program must be assumed to have created a new program whose evaluation must start afresh. The reasoning here is that such a change might introduce new faults, whose impact upon the unreliability cannot be bounded.

In such cases, we might only be able to take account of the period of failure-free working since the last failure. As one could expect, the fact that we thus use less evidence only allows us to infer less high reliability predictions. The claims that can be made *solely* from having observed a length of time *t* without failure in statistical test are quite modest: for example, under quite reasonable assumptions [Littlewood 1993], we can claim only a 50:50 chance of surviving for a *further* time *t* without failure.

It must also be remembered, of course, that all this analysis depends upon the assumption that the testing really does accurately represent the operational use of the software. Inaccuracies that cause small absolute errors in the predicted failure rate, and do not greatly affect our confidence in a prediction of, say a 10^{-3} failure rate, would seriously compromise our confidence in a prediction of, say, 10^{-6}. Even if it were practicable to observe the very long times without failures that would allow us to make strong claims for the reliability of a critical program, we would not have correspondingly strong confidence in the testing regime.

It should be stressed [Littlewood 1993], that an inability to predict a very high reliability with high confidence is not a reason for eschewing statistical testing when dealing with highly critical systems. Statistical testing is still the only way of achieving high confidence that at least the system is at least not badly unreliable. It is worth recalling that the synchronisation bug which caused the first Space Shuttle launch delay had a probability of failure of more than 1 in 100 missions: a modest amount of realistic statistical testing would give very high confidence that such bugs being absent. More recently, the Ariane V failure has again brought home forcefully the importance of realistic testing to contain risk.

5.3 Practical limitations: confidence in the operational profile

The accuracy of the reliability estimates and predictions that are computed from failure data obtained during statistical testing will depend crucially upon the similarity between the testing and operational environments. Essentially, what is needed is that the selection of inputs, and sequences of inputs, during test occur with the same probabilities as they would during real-life operation of the software. To do this, we need to know what these probabilities are, and obtaining this information can be difficult in certain circumstances. Methods for learning and using these probabilities are the topic of Section 7; we summarise here some practical difficulties.

New software poses the most obvious difficulty in estimating the operational profile, since there may be no empirical data upon which to base the estimation. With certain engineering systems, such as those that control physical plant, there may be considerable engineering expertise about this plant that can be drawn upon to provide expert judgements about the frequencies with which individual functions will be used. With systems that have a close interaction with humans, on the other hand, such as certain management information systems, judgements as to how they will be used that are made before deployment are often shown to be inaccurate, since users take advantage of the new functionality in unforeseen ways.

Even when software is not completely new, but has merely had new features added, the operational profile can change in subtle but unknown ways. Estimating the probabilities for the new features will pose similar problems to those of completely new software, above. Perhaps less obviously, the very fact of new features being added can change the probabilities by which other features are selected - some of these may increase, some may decrease.

This last observation can be particularly important in some critical systems, and may mean that extremely stringent controls must be placed upon changes. For example, in some systems, certain functions are safety critical whilst others are not. It would clearly be most important to know that changes in the non-safety-related functions had not changed for the worse the rates of safety-related failures. To do this it would be necessary to know that the part of the operational profile associated with the safety functions had not changed as a result of changes to non-critical functions. For this reason, it may sometimes be necessary to treat software following a change, *however ‘minor’*, as creating novel software for which the operational profile (and subsequent testing and reliability evaluation) must be estimated afresh.

On the other hand, good software design can do a lot to mitigate these problems. If the design is a modular one, with clear and unambiguous boundaries between modules, the problems of adding or modifying modules can be controlled better. Here object-oriented design, and software reuse can bring benefits: if a module that is to be incorporated into an existing system brings with it an extensive reliability history of similar use in other systems, it may be possible to estimate accurately the reliability of the changed system. But of course, we would need to know the operational profile of the new system over the augmented set of modules, and we would need to know that the operational profile of the module in the new system is the same as it was in the earlier systems from which its reliability data was collected. (See section 9.5 for further details)

The above remarks indicate various ways in which operational profiles can be inaccurate. Unfortunately, there is no definitive way of measuring *how accurate* a particular testing profile is, even when we later learn about the true profile from operational experience. Nevertheless, it is always prudent to collect data from operational use of a program, to compare with the profile that was used in testing and for reliability estimation and prediction. At least any gross differences here will be a warning that the reliability figures obtained from testing may not be accurate. Even an informal judgement of the accuracy of the profile used in testing, based upon evidence from the operational profile, is better than blind faith in the former.

In fact, if it is found that the operational profile differs from the one that was used in testing, it is sometimes possible to use this information to modify the reliability measures. In Section 9.5, for example, it is shown how the reliability of a modular program can be computed from the reliabilities of its component modules and the operational profile over these modules, which is defined in terms of their frequencies of invocation, sojourn times in modules, etc. If we obtain better information about the latter when we see the system in operational use, we can easily update our estimate of the system reliability.

Finally, it must be said that for many applications it does not make sense to talk of *the* operational profile: instead, there will be many different operational profiles representing the many different ways in which the system can be used. In particular, systems that are responding to human users' needs will often show different reliabilities from one user to another, arising from the different types of usage from one person to another. Examples of this effect include ubiquitous software such as word processors and spreadsheets. In such cases it may be sensible to create a profile to represent an 'average' user, and test using this; but it should be kept in mind that there may be no actual 'average' user with such a profile, and the reliabilities experienced by actual users may thus vary considerably from the one computed. The details of this problem are illustrated in Section 7.

6. Impact of adopting statistical testing on the software/system life cycle

This section explains the changes that the adoption of statistical testing may require in the software lifecycle, in terms of added activities and interactions between the parties involved. Section 8.2 explains the process of assigning reliability goals to software. This process will generally be the responsibility of reliability engineers; but an understanding of its principles will help the collaboration between these and software engineers/testers.

6.1 General

To apply statistical testing, the following is needed:

- the testers need to know the operational profile[s] determined by the intended conditions of use of the software to be tested;
- the testers or customers need to know the reliability requirements for the software under test;
- the testers need a software test and measurement harness appropriate for statistical testing.

These needs translate into actions required from both clients and suppliers of a software product. If we consider statistical testing applied to a CSCI as a whole, a typical lifecycle will be affected as follows:

- the user requirements should contain preliminary indications of the intended operational profile and of the required reliability;
- the software requirements should state more precisely the operational profile for the CSCI, in terms of explicit probabilities for the various inputs or of unambiguous indications for deriving them (e.g., specification of mission simulations). It is advisable that this be approved by the customer;
- the requirements for the CSCI should include the section "Reliability Requirements". It is the client's responsibility to ensure that the form of these requirements allows demonstration of the client's own reliability goals for the system using the CSCI; it is advisable that this section also define the decision criterion for acceptance of the software on the basis of statistical test results (see Section 9).

  In all that precedes:

  - the client may specify a classification of failures in different failure modes (e.g., failure to execute for each of the specified functions of the CSCI), and specify separate reliability requirements for each failure mode;
  - if the CSCI is meant for use in different situations, the client may specify separate operational profiles and reliability requirements for each situation.

    Including the reliability requirements and the operational profile in client-approved project documents is intended both to provide the information necessary for the successive phases and to forestall any contractual controversy, in case the software proves unreliable just because it is used with a different profile than originally specified.

- during design, the supplier may refine the testing process by:
    - allocating reliability targets (for each failure mode) to software components (e.g. HOOD objects, HOOD operations) or functions within the CSCI;
    - possibly, deriving operational profiles for said components or functions, to use in separate testing of each component or function;
- the supplier may use the reliability targets for individual software components or functions to intensify other verification activities on reliability critical components (e.g. perform additional inspections; use more stringent test coverage criteria -or require higher test coverage- apart from the statistical requirements on testing)
- a test environment must be constructed (an oracle, a test generator consistent with the operational profile, and recording tools);
- the supplier must choose (if not already specified in the requirements document) a criterion for acceptance of the software based on the reliability requirements and the statistical test results;
- the supplier must perform statistical testing. The phase in which it is performed will be:
    - for the purpose of removing bugs, at any time during development;
    - for demonstrating reliability via reliability growth models, throughout the testing and debugging process of the complete CSCI;
    - for demonstrating reliability of the CSCI in its final version, in addition to the traditional "system test" (often called "conformance testing") that aims at demonstrating, one requirement at a time, if the requirement is implemented (satisfied on at least one test case);

      The oracle can also be used effectively for conformance test; statistical testing may be applied before system testing, to collect data earlier on.
- statistical testing (with debugging and bug fixing) is monitored; observed failures and execution times are input to a statistical analysis tool, which implements the acceptance criterion;
- full regression testing after bug fixes is not strictly required for statistical testing, although it is not discouraged; it is still needed on the specific inputs that caused failures before the fix (although the results will not be used in reliability estimation), and recommended for conformance testing;
- statistical testing is stopped when the statistical analysis tool indicates that the acceptance criterion is satisfied.

Example forms of reliability requirements for a CSCI

(for a class of failures and an operational profile)

- upper bound on probability of failure per execution
- lower bound on probability of mission survival (the operational profile must include the frequency and pattern of use of the CSCI during a mission)
- lower bound on MTTF for the CSCI, in CPU execution time on specified platform

- lower bound on MTTF for the CSCI, in calendar time (the operational profile must include the frequency and pattern of use of the CSCI over time)

Note: it may be appropriate for customers and suppliers to agree beforehand on the acceptable criteria for demonstrating compliance with the requirement: the type of inference procedure that will be accepted, the confidence level in case of classical inference, and the prior probabilities in case of Bayesian inference. See Section 9 for explanations

The following checklist specifies steps needed for statistical testing, irrespective of the size of the software systems/components considered, their implementation in hardware vs software and the allocation of responsibilities between client and supplier.

Statistical testing: necessary steps

During requirements and preliminary design process:

- select measures that will be used to describe the reliability (in its general sense) of the software;
- specify numerical goals (requirements) for these measures, check for feasibility of achieving them and demonstrating them with high enough confidence;
- select level of detail (size of subsystems or functions) at which statistical testing will be applied

In the test planning phase, for each subsystem to be tested,

- assign requirements for software reliability, check for feasibility of achieving them and demonstrating them with high enough confidence;
- derive operational profile
- specify how representative test cases will be generated from the operational profile during testing
- specify oracle
- implement test harness: test generator, oracle (including instrumentation of software under test), measurement tools
- select statistical acceptance criteria and configure statistical analysis tool

During debugging and during acceptance testing:

- execute tests, under automatic control
- interpret test results to require bug fixing actions, carry out fixes, evaluate achieved reliability and/or re-tune project plans
- monitor the generated test cases to verify that they satisfy properties that are required for statistical testing to be sound (statistical distribution), or for other reasons (structural or functional coverage requirements)

At acceptance stage:

- use assessed reliability to decide on acceptance of software

During field test, commissioning, early operation:

- observations and logging facilities can be employed during operation, initially to validate the results obtained in testing, and later to obtain precise indications of how confidence in the software should grow with operational experience

6.2 Assigning reliability goals

The first step in applying a statistical approach to reliability must be the assignment of reliability targets, as explained in Section 1.3.

The principles to be applied are well established in reliability engineering. They are the same, irrespective of whether the reliability targets are established between the client and supplier of a given CSCI, or they are part of the reliability engineering process internal to either the client or supplier organisation.

Establishing reliability requirements is normally outside the responsibility of software engineers, and requires some competence in reliability engineering. This section is meant to aid software engineers and their co-operation with reliability engineers by providing: an introduction to the principles in question; some simple rules derived from applying these principles; specific caveats that apply when software is involved.

Assigning reliability goals for a software component (system/subsystem)

- Define the component in question, its boundaries and the functions it provides;
- describe (e.g., via a HAZOP) how failures of the software component affect the system[s] of which it is a part;
- enumerate relevant classes of failure modes (possibly different for different functions) that differ in terms of their consequences on the system;
- define a reliability measure (e.g., failure rate, probability of failure at a given time, probability of failure per demand) for each of these failure modes;
- model in quantitative terms how these failures affect the system;
- obtain expressions of the measures of the reliability of the system as functions of the reliability measures for the software component;
- assign bounds on the reliability measures for the component such that, if these bounds are satisfied, the system will have satisfactory reliability.

  precautions and special cases:

  software components that may be executed at the same time on separate computers: assigning to them the same reliability requirements as to the system of which they are part is wrong;

  software components that work in a redundant configuration: the model must represent the effects of redundancy, but must not assume independent failures among the components;

  software components that are not meant to interact, but such that failures may propagate between them via memory overwrite, resource contention and such unintended channels. Such components must all be represented in the same reliability model; if there is redundancy, the model must represent how failure propagation reduces the effectiveness of redundancy

  although a software system may be the responsibility of a different supplier from the hardware system on which it runs, sound reliability models may need to consider the interactions between *components* of the software and hardware systems. This is because hardware component failures may affect software failures and vice versa. E.g., hardware failures may cause overload on the remaining hardware, or because software manages recovery from hardware failures

The allocation must be *feasible,* i.e., the goals must be both *achievable* and *verifiable*. There must be reasonable confidence that the objectives can be reached and, for any system with any degree of criticality, that reasonable confidence that the objectives *have been* reached can be obtained before operation.

Defining the boundaries of a component is important to avoid either possible failures being counted more than once, or possible failures being completely neglected. For instance, it is important to decide whether failures due to the support software (e.g., operating system, file system) for an application component are counted as failures of the application component or of separate components. The reliability engineer must also be aware that if the design of a system lacks a required component, allocating reliability goals to the other components will be insufficient.

A general procedure can be described as follows:

1. establishing reliability goals for the whole system, for instance as acceptability bounds on the probabilities of failure types, on the distribution of periods of unavailability, or on the total value of losses originated by failures (see also section 2.2.5);

2. modelling the system as composed by a set of subsystems, having defined subsystems via a combination of functional considerations (e.g., following the boundaries between line replaceable units) and organisation considerations (e.g., boundaries between different subcontractors);

3. allocating reliability goals to the subsystems. This is usually a heuristic process, guided by known constraints. An initial allocation of targets to the various subsystems may be chosen, using, where possible, values that are considered feasible (on the basis of experience) to achieve and to verify. One can then verify that this assignment satisfies the system-level requirements, and tighten or loosen the requirements on individual subsystems as appropriate. Then, trade-offs may be considered, for instance if the same system-level objective can be reached with a lower reliability for some component and a higher reliability for some other component, and the savings allowed by the former change are greater than the additional expense caused by the latter.

4. possibly re-applying steps 2 and 3 to each subsystem in turn, further subdividing the subsystems into lower-level subsystems. The process continues until obtaining more detailed component reliability goals is no longer useful: component reliability requirements are only useful for allocating responsibilities within the development organisation, and as intermediate validation targets. So, for instance, it may be useless to assign a reliability requirement to a subcomponent that is produced as a part of a larger assembly, and cannot be tested without it.

For any complex system, this procedure is likely to lead to multiple constraints on the reliability of each component. For instance, in step 1 one may define separate requirements for two distinct mission phases for the system. For any component that is used in both mission phases, this will lead to two separate constraints on the reliability of the component, so that the more stringent of the two will apply. There may also be requirements based on different time scales, e.g., most alarm systems will have set upper bounds on both the probability of failure per demand and of spurious alarms per hour, often requiring two different models, leading to two sets of bounds on component reliability.

The verification phase in step 3 requires one to build and solve some model[s] of how the [un]reliabilities of the components affect system [un]reliability. This modelling may not be precise, and is typically considered acceptable if it produces estimates of system reliability that are conservative (pessimistic), as a function of subsystem reliabilities. The required modelling may be complex, but we will quote some of the more common simple cases:

- a complex control system is made up of many interacting subsystems, each of which performs a separate function, and may or may not be a digital computer. Reliability requirements are assigned to each subsystem using an appropriate reliability model for the whole system (e.g., a *series* model, if all subsystems are necessary for proper operation of the system). For those subsystem that are stored-program computers, then, one may further allocate reliability goals to hardware and software by considering that they form a series system;

- in certain cases, a whole software system is assigned a reliability requirement, e.g., in a continuously operating system, a maximum allowable failure rate, $\lambda_{r-\text{system}}$, and then reliability requirements are set on the components of the software system.

If the software executes sequentially, so that only one component is in execution at any time, one can write:

$$\lambda_{system} = \sum_{c \in Components} \lambda_c f_c$$

where f_c is the fraction of time during which component c is in execution. In particular, as all the f_c parameters are smaller than 1, a conservative allocation is possible as in the next bullet;

- if, for every component c, $\lambda_c \le \lambda_{r\text{-}system}$, then $\lambda_{system} \le \lambda_{r\text{-}system}$. Hence *assigning the same required reliability to each software components as to the whole software system is a simple, conservative criterion* for reliability allocation. This provides a rationale for the criterion used, e.g. in IEC 1508 and derived standards, in propagating required "safety integrity levels" to subsystems. Some **warnings** apply, however:
  - if software components may run at the same time on different computers[9], this criterion is no longer conservative. For instance, in a system of two pipelined components which run continuously, the failure rate of the system is the sum of the component failure rates;
  - the software components may have some redundancy. In an extreme case, they could repeatedly perform the same computation by different means, so as to mask failures that only affect some of them. Then, the overall failure rate is lower than the weighted sum indicated above;
  - this criterion may lead to setting unrealistically and unnecessarily stringent requirements on those software components that are executed very little. While being conservative is desirable, setting requirements wrongly may lead to wasting limited resources, e.g. applying much debugging effort to a seldom-used, non-critical module while spending too little effort on improving an often-used, critical module. A wrong allocation of resources may thus lead a project to produce less dependable software than it could otherwise produce;

- an alternative criterion is to allocate reliability goals so that each software component contributes equally to the probability of failure of the software system. So, unreliability targets would be assigned to be inversely proportional to the fractions of time for which the various components will be in execution. This criterion is less conservative than the previous one;

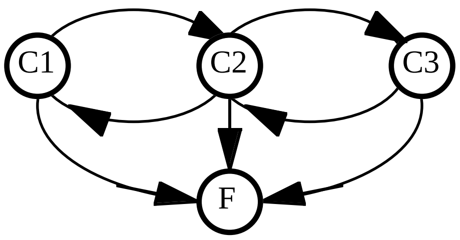
- ideally, one would allocate reliability targets to components so as to minimise the effort needed to achieve a given system reliability target. So, if a component were known to present especially serious difficulties, it would be given a relatively lower reliability target, and components that appear easier to make very reliable would be assigned more demanding targets, under the constraint that the system reliability target be achieved. A manager who knew exactly how the reliability of each module grows with the effort expended on it could allocate targets so as to minimise the project effort needed for the system to reach its required reliability. In practice, this optimisation is usually infeasible, yet it is possible for a manager to give more "lenient" targets to those modules for which achieving high reliability seems most expensive or time-consuming.

For most systems, different failure modes (i.e., forms of incorrect behaviour) exist, which differ in the character and (in particular) in the severity of their effects. This applies to failures of software systems as well. Therefore, different, separate reliability requirements will usually be appropriate with respect to different failure modes. The simplest such distinction is that usually applied between "safety critical" and non safety-critical failures, for which two different upper bounds on their probabilities are normally required. More detailed classifications are usually indicated in guidelines for safety-critical systems and software. One campaign of statistical testing

[9] This is called "parallel execution" by software engineers, but is a "series" configuration in reliability engineering jargon if the failure of any one module causes a failure of the system.

is sufficient for evaluating reliability with respect to the various failure classes, but the amounts of testing required may differ between failure classes, so that the highest amount must be chosen.

Last, a specific warning is appropriate when combinatorial reliability models are used for software components that have built-in redundancy. For instance, a safety system may use two functionally diverse modules to trigger a safety action. From the point of view of failures to initiate the safety action, the two modules are in a "parallel" reliability configuration, i.e., the safety system works properly as long as one of the two modules is working properly. This creates no problem when using statistical testing for finding and removing bugs, but it does complicate the evaluation of reliability, and thus the setting of any separate requirements for the two modules. The simplest way of deriving the probability of joint failure of both modules is to assume *independence*, i.e., setting the probability of both failing as the product of the probabilities of each one failing. However, this is wrong. In most cases, the only requirement we can establish for each of the two component modules, knowing that as a result the two-module system will be as reliable as required, is a very conservative one, i.e., equal to the requirement for the two-module system. To be less conservative, we would need to predict the *correlation* of failures of the two modules (how to do this is outside the scope of this document, but is not commonly feasible). It may then be more convenient only to set an explicit requirement (and require demonstration of its satisfaction) at the level of the two-component system.



A Markov chain for software failures in a system of 3 software components running on the same hardware. When any one of the three components fails, the system fails. States C1, C2, C3 represent execution of the three components. Transitions among them represent flow of control between components. State F represents failure.

In the more general case, more complicated models may be needed. A category of models is Markov models [Littlewood 1981, Laprie 1994]. There is a state for each subset of components that may be in execution together, plus a state for system failure. One can show that, if failure rates are much smaller than the other transition rates, the system failure rate is equal to the weighted sum of the failure rates of individual components (if executed continuously), weighted with the fractions of times that they are in execution. This model is rather general, although it is still limited by the Markov assumptions: essentially, the duration of a certain state, and the probabilities of transitions to other states, are not affected by the history of previous transitions.

Such models may be extended to consider redundancy (if the system has built-in redundancy, states with partial failure may also be needed) and the possibility of recovery. A series of such extensions is explained in [Laprie 1994]. However, which model is appropriate will depend upon the individual system of interest, and upon the degree of accuracy that is required.

6.3 Integrating statistical testing with other V & V activities

6.3.1. Direct manual selection of test cases

There may be good, special reasons for performing directed testing, in which the testers directly select test cases, even when statistical testing has been adopted. For instance, inspections may point at doubts that can be solved via specific test cases. The only precaution required is that these tests not be counted when performing inference for reliability assessment (section 9).

(A conservative criterion, to reduce the chance that the reliability prediction will err on the optimistic side because of small errors in the operational profile, would be to include *only* those, among these test cases, that produce failures)

6.3.2. Test coverage criteria

Test coverage criteria (e.g., branch coverage or statement coverage) are often mandated contractually. Whether they are a practical way of obtaining reliability has never been demonstrated in general. They would be, in a specific project, if a type of bugs were frequent enough in the type of software concerned: specifically, bugs that produce a significant contribution to the overall unreliability of the software, and are found with high probability if the pertinent code feature (e.g., branch) is executed. In addition, they may be a guarantee that rarely executed code will be checked to some extent; test cases that exercise this code, because of their rarity, might not be represented correctly in the sample used in statistical testing. Test coverage criteria are not a practical way of demonstrating reliability.

If a test coverage goal is dictated for the software under test, it can be achieved, at least in part, by statistical testing:

- the achieved test coverage should be measured during statistical testing;
- when enough test cases have been run to satisfy the criterion established for statistical testing, the coverage target may not (and often will not) be satisfied yet. Two routes are then open:
  - manually specifying additional, specific test cases to satisfy the coverage criterion. These cases must not be included in the count used for statistical inference (Section 9; but see also the conservative pessimistic criterion above);
  - continuing statistical testing until the coverage targets are satisfied. Caution is required: for stringent coverage criteria, there is a serious risk that this procedure will be prohibitively expensive, because some structural features of the software may be exercised with vanishing small probabilities.

6.3.3 Testing for rare conditions

Even if no test coverage criterion is given, there is often a concern that certain categories of rarely executed functions or code segments may have an excessive failure probability, and yet be tested very lightly, or hardly at all, by statistical testing. For instance, experience shows that this often happens with code that deals with recovery from exceptional conditions - hardware failure, data corruption. The same considerations apply here as for test coverage criteria. It should be noticed, though, that these tests are not necessarily cost-effective in terms of achieving or demonstrating reliability. As usual, whether this is the case (and only for *achieving* reliability) depends on whether directed testing of this code or these functions is more likely to find "big" faults than statistical testing. A separate consideration may be that failures of these functions are more severe than those of other functions. It is then possible to set higher reliability requirements for these functions than for the rest of the software. This will indicate the amount of testing needed for these functions, and the testers may decide to bias the testing profile accordingly.

7. Defining an operational input profile[10]

This section deals with the building of test case generators that implement the required operational profile for a given software under test. It first (7.1 through 7.3) explains the general requirements on this process, which the tester must make sure to satisfy, then describes (7.4) practical methods for test case generation, and finally deals with additions and variations which may improve the cost-effectiveness of the process. Some examples of application can be found in [Musa 1994, Musa 1993, Nikora 1994].

7.1 General description

7.1.1. Methods for specifying a profile

Ideally, a profile means an assignment of probabilities to each individual input case in the input space. For statistical testing, we want these probabilities to be equal to the probabilities in actual operation. So, there are two problems: finding out what these probabilities are; and describing them in a way that we can use in practice for selecting test cases.

Producing an operational profile

Note: these are not pure sequential steps in a process. Considerations that apply in one phase affect decisions in other phases.

Define a test run of interest (response to single demand, mission phase, time unit, ...) for each reliability requirement:

- *warning:* test runs should generally be independent samples. See later for exceptions

Define an input case:

- list the input variables:
    - explicit input variables
    - sequence, timing?
    - internal state variables?
    - platform conditions (load in a multiprogrammed system, disk occupation, ...)
- decide whether each input case must be a whole trajectory
    - are successive inputs correlated? if so, trajectory is needed.
    - decide length of trajectory, e.g. a mission, a single manoeuvre.

Determine the list of subprofiles that will be needed, on the basis of:

10 In our terminology, an input profile defines a probability distribution for the input cases. We can talk about a testing profile (i.e., a profile used in testing), an operational profile (i.e., a profile occurring in actual operation, or a testing profile mimicking it) and so on. We must warn the reader, though, that these terms are used with slightly varying meanings by different subsets of the software engineering community, and agreeing on definitions is essential to avoid misunderstandings. In particular, the term "operational" is at times used to mean that probabilities are assigned to the activations of specific "operations" (specific parts of the code) rather than of specific "functions" (defined as parts of the requirements).

- different environments for which individual reliability levels of the system are of interest:
    - different customers (organisations)
    - different users (humans or machines)
    - different mission phases or modes of operation for which separate evaluations are sought
- convenience of combining subprofile reliabilities to forecast reliability in a given environment. A convenient choice of subprofiles is one that has many subprofiles that occur unchanged (though they persist for different fractions of the total time of operation) in different environments of operation of the system.

Specify the profile (probabilities) in appropriate terms for reproducing them in testing.

The most appropriate form is decided by available information, cost and desired accuracy.

For each subprofile situation, enumerate what is known about the operational [sub-]profile, as an input for the next step

For each subprofile:

- specify the joint distribution of the input variables (do not assume independence) in explicit (e.g., probabilities) or implicit (e.g., simulator specification) form.

The possible input cases are normally so many that their probabilities cannot be described by enumerating each point in turn with its probability[11]. Three practical ways exist, and can be combined:

1. specifying probabilities as mathematical functions: for instance, for a program which reads the weight of a manufactured part on an assembly line, we may be able to specify that the weights have a normal distribution; or for a program that runs continuously for a long time and reads the time-of-day, the time-of-day input may have uniform distribution over its range. Notice, however, that for input cases made of more than one variable, the distributions of the individual variables are not enough. We need to know their joint distributions.

2. subdividing the input space into a manageably small number of subsets, and specifying a list of probabilities, one for each subset (notice, though, that this does not yet specify the probabilities of *individual cases* within each subset), or

3. instead of specifying the probabilities, specifying a process for producing the input cases, if we know such a process that we trust to select them according to the intended distribution. This process could be a pseudo-random process (a method which overlaps with method 1), a simulation of the environment based on our engineering knowledge of it, the use of test operators or of recorded input traces.

These various methods can be combined:

- at different levels of disaggregation of the input space. For instance, we can list the modes of operation for a system (method 2), and within each mode we can replay a sample of operational demands observed on a similar system in the past (method 3);

[11] When enumeration of all possible input cases is possible, exhaustive testing might also be feasible, and should thus be considered as an option, instead of statistical testing. Exhaustive testing would have the advantage of showing that the software is actually bug-free, or, more precisely, that it is so with a probability determined by the error detection coverage of the oracle.

- on different input variables. For instance, we may specify that the persons described in a database have given probabilities of belonging to one in three age groups (20 to 40, 40 to 60, and above 60), and heights which in each age group have a normal distribution with given mean and variance; or that users of a system begin their sessions with one of ten possible operations, with given probabilities, and then type an input string whose probability distribution can be emulated by reading a string with a uniformly distributed index number in a repertoire file of strings.

7.1.2. Joint probability distributions for input variables

The last example above brings up an important warning. Specifying the probability of each input case requires specifying the *joint* probability distribution for the input variables. In other words, the distributions of individual input variable, given separately ("marginal distributions") are not usually sufficient. This is due to the fact that the *value* of a variable can affect the *distribution* of another variable, i.e., in statistical terminology, that the variables are not *statistically independent.*

For instance, consider a program for which an input is the description of a hospital patient. We can specify that the proportion of males and females is 45-55, and that 10% of the patients have surnames of more than 4 letters. We would seem justified in using male patients with surnames of more than 4 letters in 4.5 % of the test cases: the two variables are independent. But let us add the information that 10 % of the patients are pregnant. This is not sufficient as an operational profile: obviously, we cannot use pregnant male patients for 4.5 % of the test cases. The test case generator must also be told that 0% of males are pregnant; or that patients that are female and pregnant are 10% of the whole.

So, dependence between input variables is the norm, and the joint distribution must be incorporated explicitly. There are many ways of doing so, for instance:

- all the input variables are produced by an environment simulator (or from a trace of previous executions), so any required statistical dependence among their values is already present (as it is caused by physical dependence mechanisms that are faithfully reproduced by the test case generator);

- "the `time_of_day` variable has a uniform distribution between 00:00 and 24:00 (minus one clock tick); the `local_temperature` variable has a Gaussian distribution and its mean and variance are given in a table for each value of the hour field in the `time_of_day` variable";

- "the reading of thermocouple A is equal to that of thermocouple B plus an error with Gaussian distribution, mean 0 and variance 5 degrees";

- "when the `mode` variable has the value `ascent`, the probabilities of commands X,Y,Z are 20%, 30%, 50%; when `mode` has the value `in_orbit`, they are 90%, 5%, 5%".

7.1.3. Discrepancies between the testing profile and the real operational profile

The profile used in testing will usually be an approximation of the real operational profile. Three separate errors will affect the generated sequence of test cases:

- the error in our knowledge of the actual profile that we wish to reproduce;

- the error introduced by these simplified ways of describing the profile for generating test input cases

- the sampling error: as we can only use a finite series of test cases, the relative frequency of any event generated in the test will not usually be the same as its probability.

In practice, we can apply special caution to avoid errors in those aspects of the input distribution that we believe to be significant in determining failures.

For instance, imagine we are testing a piece of control software which has as its input a temperature, with a specified statistical distribution. However, we know that when the temperature is within a specific, narrow, range of "critical" temperatures, which happens with only 1% probability, the internal behaviour of the software is very different from anywhere else in the temperature range. Then, it is fair to imagine that the failure rate for temperatures within this range may be quite different from the failure rate elsewhere. Generating temperatures via a simple pseudorandom process may risk large errors in reliability predictions, perhaps because the pseudorandom number generator has some minor flaw which happens to give the wrong proportion of critical temperatures. Then we may wish to specify first that the temperature must be in the critical range with 1% probability, and then separately specify a (*conditional*) distribution of the temperature when it is within this range, and another distribution for when it is outside this range. Even if the pseudo-random generator is perfect, sampling error (i.e., the necessity to test with a finite number of test cases) may for instance produce no critical temperatures, and thus bias the result (we are entitled to fear this kind of error if we believe that being or not being in the critical temperature range is a major factor in the reliability of the program - even if we don't know whether its effect is to increase or to decrease reliability). Then we may wish to specify that 1% of the test cases must be in the critical temperature range. This requirement must be satisfied with care so as not to introduce other errors. If we believe failures in the critical temperature range to be more likely, or more expensive than elsewhere, we can actually bias our test generation to produce more than 1% critical temperatures. All these issues will be discussed in more detail in later sections.

7.2 How many profiles

In many cases, we may wish to test a system under different profiles, running multiple test campaigns, e.g. T_1 test runs from profile P_1, T_2 from profile P_2. These multiple profiles will typically represent:

- different environments (users, installations); or
- different mission phases or modes of operation within the same mission of the same software installation.

7.2.1. Different environments

Most systems are developed to be used repeatedly and in different circumstances. For instance, a type of plant control system will be installed in many distinct plants, which will certainly differ in their location but will also typically differ in some aspects of their construction (models or age of equipment, configuration of the plant), and their use (if the plant can run in more than one mode - e.g., producing several products, or batch production vs reloading of materials, the mix of these modes over time will vary between plants; the staff may follow different procedures, and so on). We may expect that two installations will have different reliabilities, because the distributions of input cases differ. Experience confirms this expectation. The same change may be observed if a single plant switches from one operating regime to another. Another example is that of a machine - e.g. a car - that has to be used by different operators. Drivers differ in how fast they accelerate, how often they switch gears, etc. Thus, they subject the car to different operational input distributions, causing failure probabilities to vary with the driver. In all these examples, we could say that a system, or several identical systems, may operate in different *environments*. As an example, let us consider the case of one control system installed in distinct plants. There are two main ways of looking at reliability assessment for these systems:

- considering an individual environment. In this case, each environment needs an individual prediction and thus a separate input profile;

- considering the whole set of environments.

For instance, let us consider the probability of failure per year of operation. The probability for the whole set of environments is the average of the probabilities for the individual environments (weighted with their numbers if there are multiple identical environments). Each environment has its own profile: we could define an "average" profile (one in which the probability of each input is the average probability among all the environments) and use this for testing in order to derive the probability of failure for the system. Or, we could define a separate profile for each environment, and test the system separately with each profile, to derive predictions for each environment separately. This second approach produces far more information than the former one, at a correspondingly higher cost. Either approach may be preferred, depending on the circumstances. For instance:

- if our system is a safety system, in a dangerous plant which must be individually licensed for operation, a prediction is needed for the individual environment of each plant, so the average profile is useless;

- if every failure of our system causes a known loss (e.g., an interruption in production), and we want to predict the total loss per year for all the systems deployed, we will want to predict the number of failures per year over the whole set of environments. If the failures of distinct systems were completely independent events, and unlikely, the probability of a failure over a set of n environments would be close to n times the average probability, which would thus be sufficient information;

- if a prediction is needed for a set of new instances of the system, about which we know nothing that characterises them as "non-average", we could similarly use the "average" approach to predict the number of failures per year in these new environments. This would be a best-guess prediction. If instead we wanted a conservative prediction, this could be better derived knowing the spread of failure probabilities for the individual environments, and using for instance the worst case. If we knew the new environments to be especially similar to one of the previous ones, we would prefer to know the probabilities for that specific type;

- quite often, we are interested in both kinds of measure. For instance, we would like to know both that our system has a low enough average failure rate, so that the cost of failures over many environments will be low, and that in no environment is its behaviour so much worse than average as to make it unsuitable for use in that environment.

7.2.2. Subprofiles, modes of operation: calculating reliability via factoring

There is another reason for predicting reliability separately for each one of multiple input profiles. Many systems are used under sets of clearly distinct modes of operation. For instance, a spacecraft may go through launch, orbit, re-entry, and landing. It is reasonable to assume that the failure probability of most systems in the spacecraft will vary with the mode of operation. It may then be convenient to assess reliability individually for each mode, i.e., to test the system, repetitively, with the input "subprofile" characteristic of each mode.

This brings some advantage in terms of organisation of the testing process, and greater advantages for prediction. The probability of failure of the system over a given time is the weighted average of the probabilities for the various modes, where the weights are the fractions of time spent in each mode (see Section 9 for details). This manner of calculation may allows us to:

- compute the reliability of the system in any new environment, without new tests. It is sufficient to know: i) that the reliability in a given mode does not change between the two environments, and ii) the fractions of time spent in the various modes in the new environment;

then these fractions can be used as weighting factors in a weighted sum of the reliabilities relative to individual subprofiles;

- assess the range of reliability that may be observed in a new environment, if we know i) but not ii) (in the previous bullet point);
- reduce the number of tests needed for a certain confidence in our prediction (see later, "stratified sampling");
- refine predictions by considering that failures may have different severity depending on the mode of operation.

To define a profile as a set of subprofiles, we need to specify, for each situation we define as having a distinct subprofile: i) its probability, and ii) the probabilities of the individual input cases within that subprofile. For instance, if we define subprofiles as corresponding to different modes of operation of the software, the probability of a given input case, *v*, in the complete profile for a given system in a given environment, is given by:

$$P(\text{input case} = v) = \sum_{m \in \text{Set of all modes}} P(\text{input case} = v \mid \text{system in mode m})\, P(\text{system in mode m})$$

here the first multiplicand inside the summation is the input distribution, *conditional* on the mode of operation. The second is the probability of the system being in a certain mode (or, in general, of the input case belonging to each subprofile). This second distribution will be used to select a subprofile for each new test run (or to pre-assign the numbers of test cases to be drawn from each subprofile). The test case generator will then be driven, for each given subprofile, by the first distribution.

Some special cases are those in which a subprofile is defined as including all runs that use a certain function of the software, or all runs that use a certain module inside the software, or a certain subset of the input space[12]. If these runs are legitimate test runs (cf next subsection), such a subdivision into subdomains simplifies both the generation of test cases and the process of predicting reliability from the test results.

7.3 Definition of test runs, independence of samples: effects on reliability prediction

We examine here some necessary precautions in setting up test case generation and test measurement, for the purpose of reliability prediction.

7.3.1. Introduction

As a background for this section, we recall that, as mentioned earlier, there are two typical regimes for generating test runs:

[12] In a version of this case, often called *binning* of the input space, these subsets in the input space (*subdomains*) are a *partition* on the input space, i.e., they are non-overlapping and their union equals the whole input space. A subprofile (bin) is then defined as assigning probability 0 to all test cases outside a specific bin. In the simplest case, the partition is defined in terms of ranges of a variable, for instance, runs where the variable is positive, negative or zero.
Pre-assigning the sample size for each subdomain (making it proportional to the probability of that subdomain, instead of letting it be determined by random sampling from the whole operational profile), is often recommended. It avoids the risk that, in a specific test campaign, small but potentially failure-prone subdomains receive too few test runs. If more is known about the failure-proneness of individual subdomains, more effective allocation strategies are possible (see section 7.6).

- at each test run, the software under test and the test run generator are started anew and run until a natural unit of execution (e.g., an execution of a batch program, a mission or mission phase) terminates, or until the software fails. Sampling from the operational profile is obtained by choosing the conditions for each run. This regime, when practical, is the least problematic for achieving trustworthy reliability predictions;

- the software under test and test run generator are started and then left running for a long time (for one or more times in a test campaign). Test runs are defined as natural units of execution during this long execution sequence: e.g., responses to individual commands, transactions, frames of repetitive execution. In the limiting case in which the software is assumed to fail at constant failure rate in continuous time, a test run could be said to have infinitesimal length. Sampling from the operational profile consists in an appropriate selection of test runs from within the long sequence[s] of execution. This regime is often the only practical regime to guarantee that the test cases have a realistic distribution, both in the values of the internal variables of the software and in the sequences of values (trajectories).

There are a few forms of reliability measurement that are typically sought, corresponding to different views of the software:

- a Pfd (probability of failure per demand). The software is seen as being subject, in operation, to statistically independent demands (e.g. a launch, a flight), such that all have the same Pfd (or, possibly, all demands within each given *category* of demands - subdomain, subprofile - have the same Pfd);

- a failure rate over a continuous time scale. The software is seen as executing continuously and failing with a constant failure rate, independent of previous failures (Poisson process), although one may have to discriminate between different operation regimes (subprofiles) with different failure rates that must be estimated separately (e.g. process control in continuously operating plant);

- a failure rate over a discrete time scale (e.g., number of manoeuvres from the beginning of the mission). This is an intermediate case.

This section deals with specifying the length of test runs and how they should be sampled. In all cases, the total duration of time or number of test runs required for acceptance of the software is determined by the reliability estimation procedures, explained in Section 9. The guidelines in this section are needed to ensure that the input to those procedures is appropriate. A frequent concern is with some form of statistical *independence* between the successive obsevtaions of the software.

7.3.2. Typical recommendations for test run definitions

Some typical examples of test run definitions are as follows.

- for a test regime in which the software and test case generator are restarted at each test run and each test run is executed until its natural end:
  - mission-critical software that operates for a limited amount of time (mission, mission phase), with strong correlation among its successive operations. Requirements should be set on the probability of surviving through a mission, or mission phase if phases can be considered independent. A test run should be a whole mission or mission phase, and the measure sought will be a Pfd;
  - a situation as above, but such that testing by whole missions (or mission phases) is impractical. One reason for this may be excessive length. Another one is as follows: there is known to be a set of possible mission types (e.g. different payloads, or launches in different season, or at different latitudes), and it is desired to test each type at least once (this is not a statistical requirement, but it may be a sound requirement on other grounds), and the required statistical sample of whole missions is too small achieve this.

It is then useful to try and subdivide the mission into smaller phases that both allow independent sampling in testing and can be used as "building blocks" for predicting mission survival. The measure sought will be a Pfd (or Pfds for individual phases). In the case of missions of indefinite or very long duration, the next case usually applies;

- for a test regime in which the software and test case generator are run for one or more long sequences of execution (trajectories of input cases):
  - software for which well defined "demands" and corresponding test runs can be defined (e.g., manoeuvres for a spacecraft control system, telephone calls for a switch). A Pfd measure is sought, but organising independent test runs with re-initialisation each time to obtain a representative test case is impractical. So, the software is tested along a trajectory of test cases. This, however, causes non-independence between successive runs. Solutions are: sampling test runs that are far apart in the execution sequence; or making sure that test runs can be shown to be independent by considering the internal operation of the software;
  - software that will operate over a long, possibly unknown period of time and is believed to fail with constant failure rate (Poisson process). For instance, software that will control a satellite in orbit, for which a minimum duration of operational life is planned but a much longer duration is contemplated. Testing is directed at estimating a failure rate (over the proper time measure: CPU time, calendar time, number of frames of execution, number of manoeuvres...), so that the probability of survival can be extrapolated for any required duration.

    The duration of test sequences is determined as a compromise: short test runs are expensive because they require frequent re-initialisations (with suitably random values of all variables). Long test sequences mean that only few sequences may be performed. This may in certain cases cause large errors in the estimated reliability due to sampling effects. This leads to the next case of interest;
  - an important case is that in which the failure rate is constant over a mission, but some missions have a high failure rate and others have a low failure rate, and we do not know which factors determine to which class a mission belongs[13]. Then, running few, long test sequences makes it more likely that they will all, by chance, come from the same category of missions and thus produce a biased reliability estimate. In practice, we wish to run enough "different" missions for the various failure rates to be represented in our sample, so that an average failure rate can be estimated;
  - a general way of allowing shorter (and thus more numerous) test runs is to make sure that the software's operation contains "renewal points", i.e., events such that after each renewal point, the software's behaviour is described by a repetition of the same stochastic process. One way of achieving this is to reset the software, provided that the series of input cases in the new process is independent of the previous series.

7.3.3. More complex cases

There are more complex cases that may require ad-hoc treatment. The details are outside the scope of this document. They are mentioned since an ad-hoc definition of test case generation procedure may well be worth the necessary effort for a critical system or for a whole class of software. Such cases include:

- in the case of estimating the failure rate for software running over long periods of time, additional constraints apply if the failure rate varies over time (i.e., failures are not a simple Poisson process). For instance, the failure rate might change with astronomical phenomena, or

[13] If we knew these factors and thus had a clear classification of mission types, we should represent the mission types as different subprofiles, allowing us both to estimate an average failure rate and the failure rate for a mission of a specific type.

with the ageing of the spacecraft. Then, all runs must be long enough to allow for this variation in failure rate to manifest itself. Alternatively, the tester may be able to represent the factors that cause the variation of failure rate as explicit values (or distributions of values) for input variables, defining different subprofiles;

- failures may simply tend to be clustered in time more than a constant failure rate would imply (this is another way of saying that there is dependence among successive failures, or that the failure process is not a Poisson process). Then, to extrapolate the reliability of the software over an arbitrary period of time, a single number, like an MTTF, would not be enough. There are ways (e.g. via Markov chains) of modelling the software's behaviour as alternating between periods with high failure rates and others with low failure rates. The modelling would also indicate how long test runs have to be.

  A simple case is one in which the failure process can be seen as a sequence of failure bursts, in which the occurrence of the bursts themselves is determined by a Poisson process. For instance, consider a continuously operating, cyclic control program. A natural unit of execution is a time frame in the periodic schedule. After failing during one frame, the software is very likely to exhibit further failures (see later for possible causes), i.e., a failure burst. This makes the process non-Poisson, and creates some problems with reliability prediction. However we may instead consider a whole failure burst as the event of interest. We thus turn to predicting the rate with which new failure *bursts* (i.e., the first failure in the next burst) will occur, and if burst happen far apart over time the prediction procedures from Section 9 are appropriate (the process of "burst starts" is well approximated by a Poisson process)[14].

7.3.4. Reasons for lack of independence between runs

For many simple statistical inference methods to apply, statistical independence is required between the outcomes of successive test runs, i.e.:

P(i-th test run fails | (i-1)-th test run failed) = P(i-th test run fails)

There are many reasons why this may not hold:

- the software has internal state variables which are not reset between runs: the first failure corrupts this state, or is an indication of an already corrupted state. This increases the probability that successive runs produce failures as well;

- failures are more likely in overload conditions: a failure implies increased probability that the computer is in overload, making a second failure more likely;

- a test run is a single frame of execution of a real-time control software system executing with a simulator of the environment. Many input variables and state variables of such software represent the physical state of the environment, and describe quasi-continuous trajectories in the input space of the software. If the failure points form connected failure regions in the input space, a first failure indicates that the input trajectory has entered a failure region, and a failure at the next run is thus more likely;

[14] The difference is relevant, and which measure is of interest depends on the actual requirements. Imagine a spaceborne program with a long mission, for which a one-frame failure means missing one operation (say, taking a photograph) which brings a certain revenue. Then, I want the average failure rate for actual failures, to know how much loss I am likely to incur (while a Poisson process is completekly described by the failure rate only, for a bursty process one needs more parameters). If, instead, the first failure of the program is mission-critical, reasoning in terms of time to the first burst (i.e., the first failure in a burst) is appropriate, to predict the probability that the spacecraft will complete a mission, or last a given length of time. Last, if the spacecraft can survive a short failure burst (the program can recover, and the inertia of the craft will prevent a short improper output from causing physical damage), both the frequency and duration of bursts are of interest.

- considering again real-time control software, we must consider that some regions of the input space are more likely to contain failure points than others (for instance, because they correspond to situations that are more difficult for the programmers to treat correctly). A first failure is an indication that the trajectory of input cases is likely to have entered such a region, hence increasing the probability that the next run will also be a failure.

All these reasons point to *positive correlation* between successive failures:

P(i-th test run fails | (i-1)-th test run failed) > P(i-th test run fails)

In order to avoid this, we would wish to make any two test runs "completely" independent, in the sense that all variables that may affect the outcome of a run (success or failure) must take statistically independent values between the two runs. This is, in a sense, overkill; the reason we need it is that we do not know *which* factors specifically affect the probability of failure in this specific software product (because we do not know which defects it contains).

Seeking independence among test runs that have been defined arbitrarily may lead to inconvenient requirements. To achieve independence we should ideally remove the causes of correlation. Internal variables should be reset between runs, the computing platform should be kept shielded against external influences. Furthermore, input cases should not be selected along a trajectory. But this would run against the other requirement that they (including the initial values of internal variables) should be representative of input cases that occur from executing along trajectories, a requirement which would become very expensive to achieve.

However, more practical solutions are often the ones indicated previously. In all these examples, positive correlation will also exist during real operation of the software. When seeking a Pfd, the solution is to choose as a test run a period of execution such that independence will hold between successive runs during real operation. Then, if test runs are run under the same conditions as real operation, independence will be guaranteed. When seeking a failure rate, the solution may be to consider the burst occurrence rate (statistics of the time to the first failure in a burst).

7.3.5. Effects of non-independent sampling on prediction

Assuming independence between the outcomes (failure or success) of successive runs in conditions where this does not hold produces various errors. Although this chapter is about test case generation, it is appropriate to point out errors that may be caused in establishing requirements, and in matching the definition of a test run to the requirements [Chen 1996, Strigini 1996].

- *positive correlation in testing:* positive correlation among outcomes of test runs produces *optimistic* estimates of the probability of failure per run in operation;

- *positive correlation in operation:* positive correlation among outcomes of runs in operation causes estimates of reliability (probability of failure-free operation for a given time) that are based on correct probabilities of failure per run, *but* assume independence between runs, to err on the *pessimistic* side; in this case, it is often desirable to redefine a run so as to make runs independent. For instance, to derive the probability that a spacecraft control system will survive a certain mission phase, it may be appropriate to make a test run reproduce the whole mission phase. Reliability requirements should be stated as a probability of surviving the mission phase, rather than as a probability of failure per frame of execution or an MTTF;

- *positive correlation in both test and operation:* if test runs are positively correlated, and one uses test results to predict probability of failure per run and thence reliability in operation where runs are positively correlated, the resulting prediction may err in either direction. The solution is again to choose as test runs more appropriate slices of execution.

These statements are worded in terms of discrete test runs and hence of a Pfd or failure rate over discrete time. Similar statements apply to continuous time, substituting "Poisson process" for "independent runs".

7.3.6. Examples of mitigating factors allowing simplified specification of test runs and operational profiles

An example will illustrate the special cases in which some of the requirements described above can be relaxed[15].

A telephone switch treats long sequences of calls. A natural way of testing a switch is to submit it to a long period of real or simulated traffic. Since the switch has a huge internal state (calls in progress, billing information, state of the various lines and hardware resources,..), setting it up with a realistic, randomly chosen state for each test run may be time-consuming, and actually the best way of reaching a "realistic" internal state may be to run the switch under load for a while. Calls are a natural unit of operation (so one may want to derive a probability of failure per call), but they are not independent events, for various reasons:

- each call changes the state of the switch and related databases (with both short-lived state data, like the indication of which lines are busy with the conversation, and long-lived data, like billing information for the parties in the phone call). So, each call affects the behaviour of the switch at the next call;

- special circumstances will cause special patterns of calls. For instance, an event in a certain area may cause calls to that area to become more frequent than normal. This means that in the distribution of all calls, there may be a positive correlation between the destinations of calls that occur within a certain period (a call to a given destination increases the probability of further calls soon taking place to the same destination);

- switches located in different geographical areas will have very different input distributions: for instance, the most frequently called area codes will be different, the load patterns may be different, and so on;

- calls may actually cause each other: e.g., after a call from person X to person Y, a call from Y to a third party may be more likely than without the previous call.

So, in general, two calls cannot be two independent test runs, and a sample of calls from a long stretch of continuous running is not a sample of independent calls. It seems that we would need to test the switch over many periods of time, and each period must be a true random trial from the distribution of all possible periods in all possible geographic locations of switches. Testers are able to temper these severe requirements by several considerations:

a. although each call alters the state of the database, it seems unlikely that the specific characteristics of one call (say, which number was called) will greatly affect the chance of failure on the next call. So, the sequence of calls during an actual or simulated period of operation could be considered as independent samples;

b. even if the above is not strictly true, most of the effects of a phone call that are likely to change the failure probabilities of a future call will be short-term. So, in practice, two calls that are far apart in time, or two one-second intervals of operation far apart in time, will exhibit practically independent failure behaviour;

[15] The appendix "Mathematical definitions of important special cases of stochastic processes" summarises some of the formal definitions of special properties that are exemplified here.

c. the testers choose to represent the special patterns of calls, like the temporary high fan-in of calls to a certain area, by separate sub-profiles or operating modes. The assumption is that the special distribution of calls to this area is described with reasonable accuracy by stating that calls to this area are a larger faction of the total than usual, but, conditional on being addressed to this area, their other characteristics (destination, call duration, ..) are a random, independent sample from the usual distribution. By specifying enough subprofiles (day, night, different overload scenarios,..) the testers expect they can obtain a good sample of the overall profile;

d. although switches will be deployed in many different circumstances, differences like the local area code will not affect their failure behaviour. So, whole sets of locations can be represented by a few different subprofiles, with different values of the hardware configuration, load situations, and such.

In the end, these simplifications amount to a huge reduction in the testing effort. Whether the test operation is "packaged" in small or long runs does not change much the total operation time required for a certain predicted reliability, but it does change the overhead of organising the runs, setting the initial state of the switch, deciding the number of subprofiles and specifying their characteristics and relative frequency (with long stretches of operation, many short-term variations in load patterns will be manifested without the need for the testers to specify them as separate load conditions - subprofiles - with assigned probabilities).

Clearly, all the considerations that allow us to simplify the testing are assumptions on the actual behaviour of the switch, which must be based on fact. Some of them can be tested experimentally; others can possibly be verified with sufficient confidence by inspecting the design of the switch (e.g., that the effects of a call on a certain data structure are completely overwritten at the end of the call). Quite often, these assumptions are made without explicit consideration, and this may be misleading. It is also useful to consider possible exceptions to such assumptions: for instance, assumption d, above, may have an exception in that switches in a 2-digit area code may have different behaviour from those in a 3-digit area code. Some examples of practical application can be found in [Musa 1994, Musa 1993, Nikora 1994].

7.4 Learning and replaying a profile

Various methods exist for generating test cases according to a given operational profile. Of course, the most accurate way may be operating the software in its complete, real environment (e.g., flight test of aircraft), but this is usually far more expensive than the other methods available.

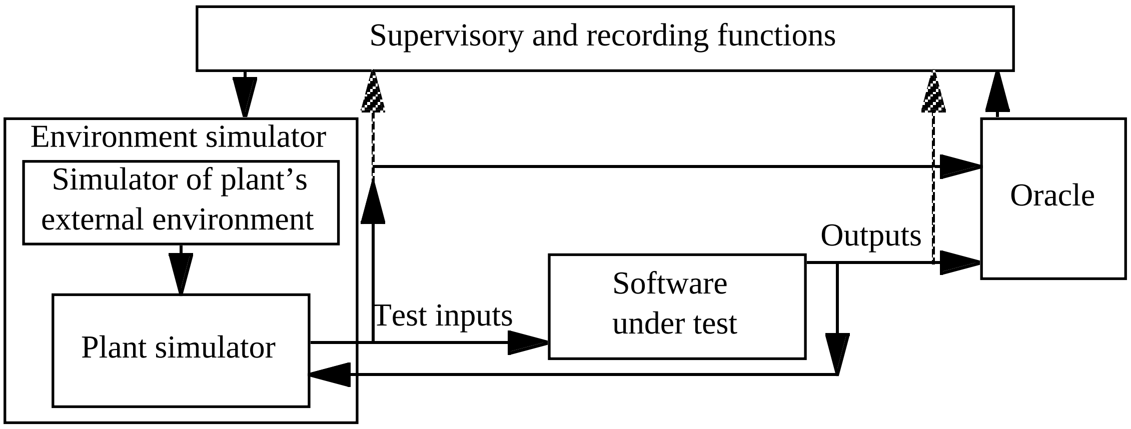
We describe a method that is especially suitable for real-time process control, and then alternative methods. The choice is a matter of practicality: available information about the operational profile, ease and cost of applying the method with the specific software under test (including considerations on the reusability of test case generation tools), and confidence in the accuracy of the results obtained.

7.4.1. Simulation of the physical system

In this approach, the software under test interacts with a simulation of its intended environment, e.g., it "controls" a simulator of a plant. The simulator is made to mimic appropriately the real behaviour of the environment by:

- reproducing the parts of the behaviour that are dictated by known physical laws;
- feeding it the right statistical distribution of values for those variables that are dictated by apparently random processes.

For instance, one can build a simulator of a missile to react to outputs of control software according to the known physical laws that govern the real missile's reaction to commands from its control system, and could supply perturbations from the environment external to the missile (e.g., turbulence, wind direction and speed) according to measured distributions of these variables.



Structure of a test harness using an environment simulator

In order to build a simulator, one needs knowledge of how the environment evolves and reacts to the software's outputs. In the case of the missile, this knowledge takes the form of known laws of ballistics, rocket combustion and so on.

The advantages of testing with simulators include: i) one can test the first generation of control systems for a new plant; ii) if the simulator is correct, it will produce input trajectories that the plant will produce *when controlled by the new software*, and iii) the simulator amounts to an automated oracle with respect to the actual requirements - e.g., "keep temperature within given bounds", or "limit unintended roll oscillations below given frequency and amplitude values" - rather than the software design specifications. Confidence in the results is conditioned, however, by the fact that i) the simulator will never exactly reproduce the behaviour of the real plant, and the differences might systematically bias the statistics of the generated trajectories; ii) the simulator may not reproduce some factor that affects the behaviour of the software[16]. Simulating the behaviour of human operators "in the loop" may present special difficulties.

This last point is best described by example. Suppose that the specification includes "don't care" indications for what must be done in certain situations, and that, in the actual implementation of the software, what happens in these situations depends on the state of variables that are unobservable or unpredictable, but are affected by the differences between the simulator and the real plant. For instance, consider a "non-deterministic" message receive statement (like a "select"): if more than one message has arrived, the executing task is handed one of them as an input. Suppose that the simulator runs slower than real time: messages triggered by events in the environment will be created less frequently than in reality, the "select" statement will more often have only one message to choose from, and this one message may be different from the message that would be chosen in a real-time execution.

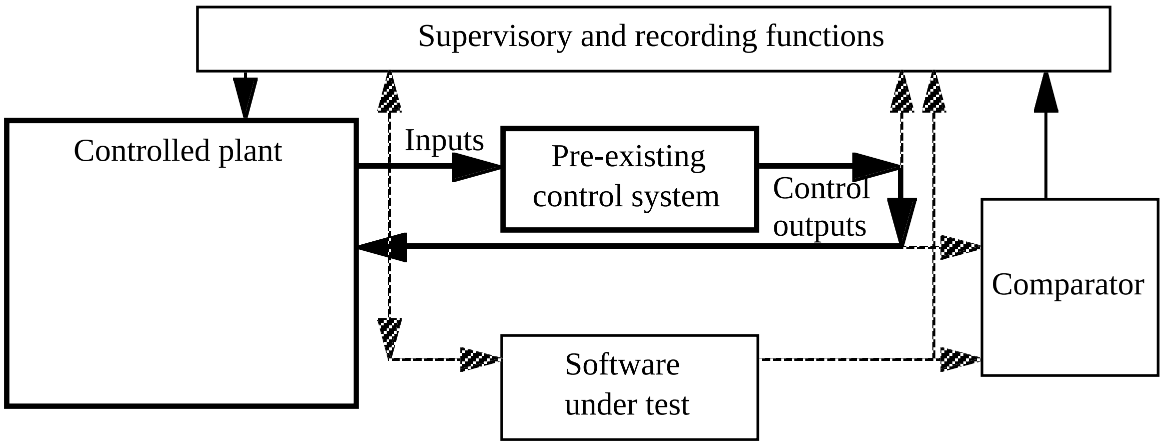
7.4.2. Using a direct sample from the actual process

If the system under test is a new version of a system that is already in operation in the intended environment, the best way of reproducing an input profile may be to record input traces in the operational environment itself. In this case, one never needs a description of the profile, as the traces already represents samples drawn from it.

16 Building a simulator is in some ways similar to building the software itself. Certainly, an incomplete or incorrect understanding of the physical behaviour of the controlled system (or of the requirements on its behaviour, e.g. the specification of a safe "envelope" for it) may produce faults in the simulator as well as in the requirements for the control software. This may have serious consequences: the worst case would be that tests based on the simulator simply could *not* detect a particular fault of the software.

We may call this the "in-plant, back-to-back" method of testing. Consider for instance a control system for some industrial plant. If the plant is already in operation with a previous-generation control system, the new control software can be run with sequences of sensor inputs as generated from the plant itself. The new control software may run on-line, in parallel with the operational control system, or off-line with recorded data. In either case, this method has the advantages that, if the specified behaviours for the old and the new control systems are similar, the input trajectories are certainly realistic and (tautologically) statistically representative of the operational situation where the testing is done. In addition, the old control system provides a built-in automated oracle for deciding - by comparison - which outputs from the software under test are correct. In practice, one is testing the new software "back-to-back" with the old control system, investigating each detected discrepancy in the outputs as possibly due to a fault.

This kind of testing requires recording equipment, or the possibility of connecting the new software in parallel with the old one.



In-plant, back-to-back testing, with the new and old control systems on-line in parallel

Possible problems include:

- the new system may behave differently from the old system. First, there may be slight differences within the limits of the specified correct behaviour; second, the specifications of correct behaviour for the two systems may be somewhat different: for instance, the new system may aim at more efficient control, or have the capability for altogether new control modes, compared to the old system. This makes it impossible to use the old system as an oracle. The advantages of realism and statistical representativeness may also fade, as the new software would cause the plant to exhibit a new, different statistical distribution of trajectories. The same problem may happen if the system is interactive, and it provides features that were unavailable in the old system and cause users to substantially change their procedures;

- the data available from operation are not enough for the amount of testing that is required. For instance, we may be testing a safety shutdown system for an existing plant, where we know that the trigger conditions for a shutdown have occurred ten times since the plant has been operational. Then, we only have ten realistic test cases, which are insufficient for reaching any high level of confidence in the new software. We must then adopt a different method for generating test cases. The ten realistic cases can and usually should be applied to the new software, just as additional insurance against the possibility that the test generation process we employ lacks some essential characteristic of the real environment.

7.4.3. Pseudorandom generators

Pseudorandom generators can be used to generate numbers with a specified statistical distribution. The distribution of each input variable can be obtained by observation of actual profiles, general physical knowledge and/or a priori considerations. For instance:

- many physical variables are known to have a Gaussian ("normal") distribution (a priori information; possibly to be confirmed through observation);

- the distribution of wind speeds at a given altitude can be obtained from general meteorological knowledge, without reference to the specific system that will be affected by the wind;

- the probabilities of the various commands issued to a spacecraft can be obtained from their relative frequencies, measured on logs of past missions; the distribution of the values of a parameter to a command (e.g., duration of firing of a rocket) can be obtained from the same logs, via statistical inference.

7.4.4. Simulation by Markov chains and other time-related modelling languages

Faithfully simulating the behaviour of an environment may be difficult, either because the laws that determine it are insufficiently known or because the simulation would be too expensive. For instance testing an autopilot for boats may require a description of the waves that disturb the heading of the boat. We might describe waves in terms of the deterministic behaviour of particles of water interacting under the effect of gravity, but, for the sake of economy, we may well prefer to describe their heights as generated by some pseudorandom process, and just ensure that the process resembles the real sequence of waves in some essential characteristics, like the general statistical distribution of heights. Since we know that the heights of a sequence of waves follow an essentially chaotic pattern, we can consider this form of simulation as satisfactory.

There are many other cases in which we may wish to mimic the behaviour of an environment through a pseudorandom process, which resembles the real process in some important measures. However, the tester always needs to decide whether the aspects neglected in this simplified model are important for the realism of the testing.

Markov chains are a language for representing processes that develop in time. The nodes represent *states*, and the edges represent *transitions* between states. A Markov chain gives a compact representation of a great number of possible behaviours, and can thus greatly simplify the simulation of an environment. A model driven by a Markov chain differs from a simple random number generator in that it reproduces, to some extent, the "memory" inherent in real physical processes, in which the probabilities of different future states depend from the current state. However, it is a simplification of the actual behaviour of the environment. Essentially, a Markov chain can be built so that its behaviour will be identical, as far as certain statistical measures are concerned, to that of the real environment.

There are several possible ways of using Markov chains as part of an input generator. We give a few practical examples. For instance, a Markov chain could be used to generate, by simulation, sequences of:

- load states on a telephone switch, where a load state is specified by the number of calls in progress for each type of call that the switch can process [Avritzer 1995];

- executions of the various modules in a piece of sequential-execution software, following the flow of control through call and return operations;

- buttons pressed by an operator, on a control panel with push-buttons only

- states, modes or user actions in an interactive system [Whittaker 1994]. E.g., the use of an editor may evolve through the states: input text, search for text, delete text, spell-check, etc. A separate method must then be used to generate the details of what the user does in each state or task (e.g., the strings of characters they input as text).

A disadvantage of Markov chains is that they cannot easily represent processes where the probabilities of transitions to the possible next states depend on the states that preceded the current one. This is often the case for meaningful sequences in the environment that we need to model. We can often deal with it by making the chain more complex, e.g., with additional states that distinguish between different sequences (e.g., we might substitute the one state "the brake pedal is fully depressed" with two different states: "the brake pedal is fully depressed because the car is standing still at a traffic light" and "the brake pedal is fully depressed because the driver needs to decelerate quickly"), but this process makes the Markov chain more complex and may thus lose its main advantage over a full-fledged simulator.

Other languages than Markov chains are available for specifying a pseudo-random process so that some of its statistical properties are similar to those of the actual process. Queuing networks and stochastic Petri nets are two examples.

7.4.5. The problem of representing the timing of inputs

The sequence of inputs given to a computer system determines the software's behaviour by the timing of the inputs as well as by their values. Without accurate representation of the timing, the input sequences that are generated might well be not just unrepresentative of the intended profile, but actually impossible.

When replaying recorded input traces, or using a simulator, this is no problem. When, instead, sequences of inputs are generated synthetically by a pseudo-random process, timing must be represented somehow. A time-triggered software design (one in which all sensors are read periodically, and all scheduling is periodic, without interrupt-driven activities) makes this easy: the timing is implicit in the cyclic sampling performed by the software. "Simulated time" in testing can even be speeded up or slowed down, without affecting most aspects of the behaviour of the software. In more event-triggered designs (relying more on interrupts from sensors and on dynamic scheduling), instead, the input generation process must be driven by a more complex specification, indicating *when* the next input will be presented on a certain variable (e.g., an interrupt will be produced), *which* variable will be affected, and *which value* will the variable take. The failure behaviour of the software may be affected by many details of the timing (via the reactions to interrupts, the duration for which interrupts may be disabled or may overrun previous ones, the scheduling algorithm, the policy in choosing between concurrent invocations to a module, etc.), making it more difficult to trust that a test generation process, short of full-fledged use in the target system, is statistically representative.

7.4.6. The problem of input from human users

Human users of interactive system are, in principle, a part of the environment as all others. The ways of learning and reproducing human behaviour may include actual recorded traces (on previous systems, or on prototypes or simulators of the new system), interviews with expert operators to determine which action plans they would choose as a function of the situation, and direct use of the system by an appropriate sample of users, selected by taking into account the physical characteristics (e.g., reflexes, visual acuity), job roles, level of training and expertise, of the actual intended user population.

It is also possible, to some extent, to simulate human users of interactive systems. Users go through sequence of operations. For instance, a person using an editor program may insert a sentence, then move to some other place in the text, delete text, etc. After each step, there are many alternatives for the next step (although at each step in the sequence there may be some operations that are impossible: e.g., "undo" operations are only allowed after a change to the text). This behaviour may be modelled by a Markov chain, for instance, or by a more complex simulator taking into account the possible intentions of users.

Describing and reproducing the users' contribution to operational profiles may present specific problems:

- human behaviour may present large variations between individuals. Even with software for which simple usage patterns are prescribed (e.g., software for maintaining or configuring systems), different users may have different ways of accomplishing the same goal, ways of grouping batches of operations into larger or smaller "jobs", and patterns of errors (both slips, like pressing the key next to the chosen one, and conceptual mistakes) in using the software. For software that allows more complex command patterns, like a spreadsheet program or computerised controls in an aircraft, users may well have distinctive styles and/or goals, leading to command sequences that are very different and not easy to describe as generated by a statistical distribution;

- depending on recorded behaviour of the users of an earlier system may lack realism: differences between the older and the newer system (e.g., the possibility of obtaining a certain result via a different command sequence than before, changes in the relative ease of use or performance of two allowed command sequences for the same goal) may well cause radical changes in user behaviour. Even if the system has new functions which were absent in previous generation systems, however, it may be that these correspond to actions that were previously performed and logged manually, so that the logs can be used to reconstruct profiles of operations required, though not of the detailed command sequences that operators will use;

- actually using human operators to interact with the software under test may be slow, and the difficulty of obtaining samples of users clearly varies: a sample of computer game fans could be obtained for free, a sample of novice secretaries at some expense, and a sample of high-level company executives is usually prohibitively expensive. A special case is that of expensive personnel who receives training in simulators, as simulator time may provide inexpensive, reasonably realistic input for software testing.
  Users working on prototypes can be used for learning a profile, or for double-checking a profile produced via other means. This possibility is particularly attractive when the system to be tested is radically new and/or offers radically new functions. Trust in prototypes must be limited by the fact that the way they are built may skew the users' behaviour. For instance, a very convenient function, which will be used heavily in the finished system, may not yet be implemented, or may be implement in ways that discourage it use (e.g. with a cumbersome human interface, or inefficient implementation).

7.5 Verifying that a synthetic profile is representative of the intended distributions of key variables

When test cases have been generated by sampling from a distribution - rather than, for example, recording actual usage of a program - it is important to check that the test cases truly are representative of the distribution from which they come. There are many standard statistical tests[17] that can be used for this purpose, by examining the *goodness-of-fit* between observed data and the distribution from which the data have supposedly been sampled. Such checks can be applied, in principle, to any measure that characterises the test runs, not just the input variables. So, for instance, if it is known that a spacecraft, under given control laws, will exhibit a certain distribution in its attitude error measures, we may wish to measure the attitude errors during a test of the control software with a simulator of the spacecraft and check that these measures appear to be a sample of the known distribution: if not, we would have reason to look for defects in our test generation process (including the spacecraft simulator), which would make our statistical testing inaccurate.

For all these tests it is important to realise that what is being tested is precise and rather limited: we are examining only whether there is any evidence to suggest that the data we have observed do not

[17] The term "statistical test" (meaning a procedure for assessing whether a data set has a certain statistical property) bears an unfortunate resemblance to "statistical testing" (of software), but we will use it as it is a standard term in mathematics.

come from the distribution that we have hypothesised. The tests do not, of course, address the more important issue of whether the distribution hypothesised really is the same as the one that will apply during operational use of the software under examination.

It is beyond the scope of this document to discuss in detail the theory of statistical tests. Briefly, a statistical test uses data to examine the plausibility of a *null hypothesis*. In our case, the null hypothesis will be that the test cases come from the distribution that is believed to be that which will generate them during operational use. In its simplest case, this distribution will concern a scalar random variable. We shall restrict ourselves here to this case, but there are equivalent approaches involving vector random variables. In practice, test cases will usually be vector variables.

When the random variable is naturally discrete, taking a finite or at least countable number of possible values, or when it is convenient to group continuous data into a finite set of intervals, the commonest approach is the so-called *chi-squared* goodness-of-fit test. This compares the number of observed data points corresponding to each value of the random variable (or each interval) with the number that would be expected if the null hypothesis were true: a "significant" difference here would then be taken in practice to raise doubts about the truth of the hypothesis. Details of this approach are given in the appendix on "Representativeness 'tests'".

If the random variable is a continuous one, a chi-squared test can be performed on a suitably chosen number of intervals, but it is usually preferable to use a test which compares the *sample distribution function* $\tilde{F}(x)$ with the hypothesised distribution, $F(x)$. The Kolmogorov test is one example of a test that is based on the "distance" between these two function (see appendix): the greater this distance, the stronger the evidence that the hypothesis is false.

Whilst these tests compare a hypothesised distribution directly with its sample equivalent obtained from the observations, there are other tests that work more indirectly. Many tests compare *parameters* of the hypothesised distribution, or functions of these, with their sample equivalents. For example, in the case of the commonly-used normal (Gaussian) distribution, estimates of the first two standardised cumulants should be close to zero (see the appendix "Representativeness 'tests'").

7.6 Biasing the operational profile: stratified sampling, importance sampling

We now consider ways for obtaining more accurate reliability predictions, with a given number of test runs, by altering the test profile on the basis of known "hints" about the failure behaviour of the software. First we show a way of making the test cases used "especially representative" of the intended profile. Then we show methods whereby the tester introduces an intentional error in the way the operational profile is reproduced in test case generation, and then corrects for it when performing reliability estimation. These methods have been used in various applications of statistics to improve the confidence in the estimates derived from a given number of samples. A discussion related to software testing is in [Podgurski 1993]

7.6.1. Non-biased (proportional) stratified sampling

A practical basis for biasing the profile is to divide the input space in a set of non-overlapping subsets ("subdomains"), S_1, S_2, S_3.... Each of these subdomains, S_i, has a certain probability P_i of occurring in the operational profile,

P_i = P(randomly chosen input case is from $\in S_i$,)= $\sum_{case \in S_i} P(case)$

(such that $\sum_i P_i = 1$).

For the sampling to be representative of the actual profile, the sample size for the i-th subdomain must be proportional to P_i, i.e., out of a total of T test cases, $(T\ P_i)$ should be drawn from S_i.

As mentioned earlier, there are advantages to be gained from this method (akin to *proportional stratified sampling* [Cochran 1977]. For instance, suppose that a subdomain S_j has a small P_j, such that $(T\ P_j)$ is, say, 1. A pure random selection of test cases might accidentally contain no test case from S_j; proportional stratified selection will ensure that exactly 1 test case is taken from S_j. Imagine that S_j is a peculiar subdomain, such that most of its input cases cause failure: ignoring it would be likely to distort the results of testing, and stratified selection prevents this distortion. In practice, proportional stratification only gives subtsatntial advantages if the different subdomains have very different failure probabilities.

7.6.1. Biasing the profile

The above example can be expanded to show how it may at times be desirable to bias the profile used in testing. Imagine that S_j happens to be the only subdomain containing cases that cause failure. Suppose $P_j=10^{-2}$, and that an input case from S_j has a 0.1 probability of causing a failure. Imagine that a tester runs 1000 test cases, with proportional stratified selection, so that 10 are allocated to S_j. On average (i.e., if the 1000-case test campaign were repeated many times), this tester would observe 1 failure per 1000 test cases, which is a true representation of the software's reliability. But since the tester only runs one campaign, he has a non-negligible probability (actually, a 35% probability) of seeing no failures, by chance (and a 26% probability of observing 2 or more failures, causing an error in the pessimistic sense). Statistical procedures (see Section 9) take into account this variation by indicating a low "confidence" in the reliability estimates derived from the results of testing. But if the tester does know (an important condition!) that S_j is likely to contribute much to the unreliability of the software, he might decide to use, for instance, 100 test cases from S_j (if the budget allowed for 1000 cases, only 900 would then be left to the other subdomains), which gives a fair chance of having approximately 10 failures (for instance, the probability of seeing from 6 to 15 failures is then 90%), i.e., a correct indication for subdomain S_j.

So, the required error in the operational profile is introduced by changing the probabilities of test cases being drawn from the various subprofiles. A simple way of doing so is to assign to each subprofile a fraction of the total number of test cases, $P_i' \neq P_i$ (such that $\sum_i P_i' = 1$). In the case in which subprofiles correspond to subdomains in the input space, in practice, one may either (i) use a test case generator which can be controlled to produce test cases belonging to each S_i with the appropriate probabilities, or (ii) generate test cases from the whole profile, keep a count of those already generated for each subdomain, and discard the new test cases generated when they fall in a subdomain for which the required number has already been generated.

The reliability of the software is a weighted sum of the reliability measures for the subdomains, with the weights eual to their probabilities, P_i. See Section 9.5 for more details.

There are several reasons for wishing to bias the testing profile compared to the true operational profile. We forewarn the reader that only the first one, in the following list, usually allows practical solutions in the case of highly reliable software. These possible reasons are:

- to concentrate the testing effort on subdomains (or subprofiles, e.g., modes of operation of the plant) where failures are more critical (e.g., in a spacecraft, because they correspond to a mission phase during which erroneous manoeuvres are unrecoverable). The appropriate procedure is to set explicitly higher reliability requirements for the software when operating in this subdomain. This will automatically increase the number of test runs needed to demonstrate the required reliability for these critical subdomains;

- to get a point estimate of reliability that is more likely to be close to the true reliability (as in the numerical example given above), or a narrower two-sided confidence interval (see section 9 for details). Statistical theory shows how the number of test cases should *optimally* be allocated to subprofiles in order to minimise this risk of excessive variability in test campaign results. For a given total number of test cases, the least variance will be produced by making the P_i' proportional to the P_i and to the variance of samples from one subdomain, $\vartheta_i(1-\vartheta_i)$ (where ϑ_i is the failure probability for inputs from subdomain i). If the cost of each additional test case in subdomain i is c_i, and varies between subdomains, the least variance for a fixed money budget is obtained by making the P_i' proportional to c_i as well). Procedures for application to Markov chain-gererated inputs can be derived from [Goyal 1992];

- to get a better one-sided confidence bound on reliability. To raise the bound, or the level of confidence in a given bound, the best selection of test cases is one that "stress-tests" the software, i.e., increases the likelihood of failure as much as possible, compared to the operational profile. Of course, in order to obtain this increased confidence, one would need to estimate *how much* more likely failures are made. As this is usually unknown, there is an effect of making predictions more conservative, but by an unknown quantity;

- to improve the effectiveness of testing for improving reliability. This is theoretically possible, but difficult in practice. The expected increase in reliability is the combined effect of the likelihood of observing a failure and of the reliability improvement that is obtained by the subsequent fix. Concentrating testing on one subdomain would be certainly justified if the subdomain were known to contain individual bugs which were in turn major contributors.

A general warning applies to all these cases except the first. The information for biasing the operational profile is some kind of "hunch" about the relative failure-proneness of different subprofiles or subdomains. If the hunch is wrong, these procedures can produce more harm than good. Such hunches may come from previous experience in similar products, or the project history of different components within the software under test. A generally accepted hunch, for instance, is that high loads (for software for which there is a meaningful measure of "load") produce lower reliability than low loads. However, research has shown that many commonly accepted hunches, like indicators of "high risk" modules in a software system, can be misleading. In particular, if all the software under test was developed with a presumably high-quality process, such that very high reliability is expected, and acceptance is based on a test campaign with no failures, it may well be best not to adopt any biasing of the profile.

8. Indications for test automation and for building test oracles

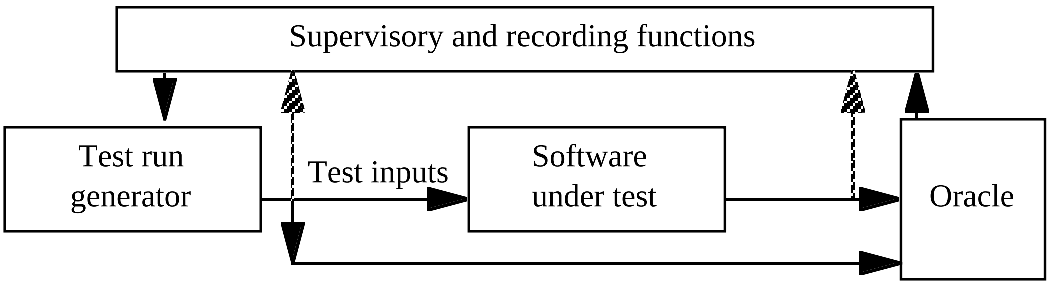
Statistical testing both facilitates and practically requires an automated testing process. Most of this section is dedicated to ways for building effctive oracles.

8.1. General considerations on test automation

The efficacy of testing, either for debugging or for gaining confidence in the tested software, clearly increases with the number of test cases that are run (although, as is well known, the rate at which faults are found with any given strategy usually decreases over time). Statistical testing, in particular, depends on running large numbers of tests: to have a reasonable chance of finding a fault in a program with failure probability *q*, or to demonstrate a failure probability of the order of *q*, we will want to run at least a small multiple of 1/*q* test cases. So, automation of the testing process is highly desirable. It is also, fortunately, conceptually and often practically simple.

Manual testing requires a human tester to select test cases that appear useful, running them through the software under test, and analysing the results for failures. This is a highly labour-intensive activity, and it makes it difficult to run high numbers of test cases, as interruptions for human intervention are time-consuming, and human effort is expensive. Automated test environments improve on this situation by combining one or more of these features: i) running multiple test runs in a single "batch" ; ii) automating the detection of failures; iii) automating (parts of) the generation of test cases. Automation allows human effort to be spent in the higher-level, supervisory activities and in actually correcting faults, and to be mostly spent in parallel with the running of test cases, rather than interrupting it. A useful feature of statistical testing is in the third area above: as we have described before, the generation of test cases can usually be specified as an automatic procedure. However, it is clear that if the other two areas are not automated as well, they will become the bottlenecks limiting the performance of the testing process.

A set up for automated testing is shown in the figure below:



The *oracle* is any mechanism used to decide whether the program behaves correctly on a given test. The oracle decides about the test outcome by analysing the behaviour of the program against its specification. Several aspects of the testing are worth recording:

- counts of test cases and observed failures (subdivided by failure mode), for assessing reliability;

- the input cases, or information sufficient for reconstructing them (e.g., seeds and sequence number in sequences of pseudo-random numbers), as well as the corresponding outputs from the software under test (or, at least information on the oracle's responses), as an aid for locating and correcting bugs. As a minimum, these ought to be recorded for those test runs on which the software is judged to have failed. However, being also able to reconstruct the test runs on which the software behaved properly may be useful for diagnosing the bugs. In addition, the recorded data can be used for regression testing, which is usually required after each fix[18].

Automated testing does not exclude the need for human supervision. A testing campaign is an expensive undertaking, and the effort may be wasted if, for instance, the test harness contains bugs. For instance, testers should analyse all suspect failures (or internal errors in variable values) as they arise, as they may well signal problems with the test harness itself, or, during the final assessment testing of a product, might signal that release is premature and the product needs further corrective work.

Indications for measurements during testing

During testing, the essential measurement requirements are:

- measuring the time variable (time, CPU time, number of test runs, ...) at each failure (or between failures);
- counting the observed failures;
- if statistical testing is continued across fixes to the software, keeping separate measurements for each successive version of the software.

Additional measurement activities that may be useful in specific cases include:

- classifying failures by failure class (i.e., maintaining separate counts for each class): necessary if separate requirements are stated for different failure classes; useful if the software under test may be reused in different environment where the severity of specific failure classes will change; and useful for improving the development process;
- classifying test cases by values of specific variables: will allow reliability estimates for specific variables ranges (subprofiles) to be derived "after the fact";
- classifying test cases by the components (within the software under test) they use: will allow reliability estimates for individual components (useful for instance as a first estimate in case of reuse), even when generating an operational profile for each specific component is impractical;
- measuring structural test coverage, by running an appropriate tool alongside the statistical test harness; useful when there is a requirement on test coverage;
- collecting statistics of values of specific input, state or output variables, which can provide a test on the statistical accuracy of the test case generation procedure;
- measuring more than one time variable: this may reveal a more "natural" way of describing the software's failure process: e.g., in terms of number of invocations or transactions rather than CPU time.

[18] Regression testing is usually motivated by a requirement to show that the step of debugging just performed did actually improve the software. It seems reasonable, indeed, to test that the input case which caused failure before the fix is handled correctly after the fix. As to the other test cases that the software handled successfully before the fix, repeating the same test sequence already performed does not, in statistical terms, give better assurance than performing any other sequence of the same number of test cases from the operational profile. Actually, if most fixes were expected (from experience) not to create additional bugs, there would be good reason for not wasting time by repeating tests that were previously run successfully. We believe that regression testing is better seen as an experiment about the confidence one can have in the ability of the debuggers to understand the software. After a fix, one would expect regression testing to show a sequence of successes and failures which only differs from the sequence obtained before the fix in the absence of the specific failure[s] the fix is supposed to avoid. If this is not so, the fix was deficient (the program is more difficult than expected for the debuggers to deal with), and this must reduce the confidence in the software that is derived from any given sequence of test results.

8.2. Oracles: specification and characterisation

The main problem with oracles is guaranteeing a high *coverage*, i.e., a high probability that a test run ending in failure will indeed be flagged as a failure.

Definition: **Coverage**

The *coverage* of an oracle is the probability that it rejects a test run, given the correct probability distribution of the inputs and given that the test run is a failure.

Indeed, if the failures that happen are not recognised as such, testing will be ineffective for improving the software, and will be misleading for the purpose of assessing the software.

One must also consider the dual problem of *false positives*, or false alarms, i.e., of an oracle that mistakenly flags a successful run as a failure. Each false alarms require human intervention to recognise that the test run was actually a success, and thus wastes expensive resources.This problem, however, if present with a small probability, may be acceptable, especially if it is a side-effect of high error coverage (for instance, by an oracle that allows slightly too tight tolerances on output values for continuous variables).

An oracle obviously needs to observe the input case given to the software under test, and, generally, the output the latter produces. However, an oracle may be made more effective by also observing the internal behaviour of the software under test. Erroneous intermediate results of the computation may become evident even during runs that either produce correct outputs or outputs that the oracle cannot recognise as erroneous. Allowing the oracle to use the internal behaviour of the software under test can be described as improving the *observability* of the latter. More details are given in a later section.

An oracle that may reject a "suspicious" test run even without observing a failure has several advantages and disadvantages[19]:

- for improving reliability, the usual considerations apply. There is clearly increased efficiency in finding bugs: a bug can be betrayed by an internal error, even in runs that do not cause failures[20]. This may lead to huge savings in the number of runs needed. On the other hand, interpreting every detected error as requiring a fix to the software undermines the confidence that bugs will tend to surface in order of importance. If the software under test has many "small" bugs, this may lead to major inefficiencies. In practice, it will be necessary to examine the specific circumstances. If the software under test is expected to be essentially bug-free, detecting errors in its internal state undermines this confidence and may well be worth investigating every time. If, on the other hand, the software is complex and presumably has many minor imperfections, it may well be appropriate to ignore internal errors that do not seem to point to important defects, at least if there is confidence in one's ability to detect actual failures;

- for assessing software reliability, test runs where errors are detected should not be interrupted, but continued until they can be adjudged as either correct runs or failures. However, there may

[19] Notice that the definition of "oracle coverage" takes on a different meaning if we design the oracle to "reject" test runs that produce internal errors but no failures. A useful definition for the purpose of comparing oracles may be: "probability that the oracle rejects a test run, given a probability distribution of the inputs and given that some variable that the oracle observes becomes erroneous in that test run".

[20] Both fault injection experiments and studies of actual bugs show that programs may have high "accidental" fault tolerance: erroneous variables may be overwritten with correct values, or be used in computations where the error is inconsequential: e.g., a Boolean variable erroneously set to "false" has no effect if it is logically ORed with another, correct value.

be a case for saving on test effort and accepting a possibly over-conservative reliability estimate.

In the next two sections, the different means for detecting errors and failures are considered, first from the point of view of their structure and then on the basis of what they check in the information they process. Ideally, a tester would use as many means as possible to improve an oracle's error coverage. In practice, compromises are always necessary to reduce the costs of producing the oracles, running tests and examining rejected test runs. Some types of checks will be excluded if they do not seem to add much to the coverage already obtainable with other types. Such decisions often depend on the assumed behaviour of failures, derived from previous experience with similar software. Clearly, these assumptions may allow large savings, but should be made with great prudence since, if wrong, they may cause decisions that make the testing largely ineffective.

Options in designing oracles

What they can observe:

- values of input, output and internal variables;
- occurrence, timing and/or sequencing of events: changes in variable values, flow-of-control events (reaching a statement, calling a procedure, sending/receiving a message...).

Means for observation:

Interface of the software under test:

- procedure call arguments and return values;
- object method call arguments, return values and visible parts of object state (via state-read methods);
- exceptions raised in the software under test;

Changes to the software under test:

- trace statements added in the software under test;
- added assertion checks producing error messages (or raising exceptions);
- changing exception handlers for exceptions raised in the software under test;
- intrusive debuggers (using breakpoints, step-by-step execution, ..);

External means:

- logic analysers and other non-intrusive debugging aids;
- monitoring of plant (or simulator) controlled by the software under test;
- concurrent audits on databases updated by the software under test

Properties we may be able to check for in the behaviour of the software under test or components thereof:

- input and outputs satisfy the relationship mandated in the specs;
- outputs , when processed via an appropriate inverse function, produce inputs;
- [output or internal] variables only take on legal values;
- [output or internal] variables only take on plausible values;
- [output or internal] variables do not exhibit unintended discontinuities in response to small perturbations of input values;
- values of variables satisfy pre-post conditions, loop invariants, etc., specified by designers;
- some behaviours or values that are specifically forbidden by the specs are never produced.

Measures of quality of oracles

Coverage $\overset{\Delta}{=}$ P(rejected | prob. distribution of inputs, improper behaviour in test run)

A high coverage is desirable; coverage is limited by:

- testing for conditions that are necessary but not sufficient for correct test runs;
- incomplete observation of the behaviour of the software under test (e.g., neglecting some output variables, neglecting the timing of outputs, neglecting changes in state variables);
- design faults in the oracle.

Probability of false alarms = P(rejected | prob. distribution of inputs, proper behaviour in test run)

A low probability of false alarms is desirable, as false alarms waste human resources. False alarms are caused by:

- over-stringent checks;
- design faults in the oracle.

Performance: usually measured as response time of the oracle, and relevant in so far as waiting the responses of the oracle reduce the test throughput, e.g. test cases per hour. The oracle's performance can be increased by:

- dropping some features of the oracle (usually accepting lower coverage or higher probability of false alarms);
- more efficient implementation of the oracle;
- running the oracle on a separate machine or set of machines (in parallel) from the software under test, and allowing test cases to proceed in parallel with the oracle's operation;
- even if the oracle and the software under test share the same multiprogrammed machine, allowing them to run concurrently without tight synchronisation.

8.3. Improving the observability of the software under test

The means available to testers for detecting errors in the execution include all means that are available for run-time error detection. As an additional advantage, testers can use these means to a greater extent than is economical during actual operation, plus others that are specific to testing.

Observing the internal state of a program is easy using debuggers, trace analysers and such common tools. In addition, run-time checking of assertions, in particular of program invariants, at least during testing, is recommended practice. It is a built-in feature in some languages and easy to implement with any language. The real difficulty is in choosing what (which variables) to observe, and specifying which conditions its values must satisfy. This will be discussed later.

In choosing error detection means, the main concerns are cost, error coverage, probability of spurious alarms (which entails a cost in human effort for recognising the alarm as spurious) and degree of interference with the operation of the software under test.

This last issue is relevant in practice because non-intrusive observation (e.g. via logic analysers) may often be much more expensive than intrusive, software-implemented observation, via assertions or debugging and tracing facilities which rely on "instrumenting" the code before compilation (adding extra statements which interact with the debug/tracing tools) and/or altering its execution (e.g. setting "breakpoints" that invoke processor traps). The distortion that intrusive observation causes in the behaviour of the software may prevent failures which would instead happen during actual operation. For instance, the added code may slow down parts of the execution and thus prevent race conditions that would instead happen without the added code; it may declare additional variables which, by creating "buffer zones" between the program's own variables, prevent overwrite errors from causing detectable problems; etc.

There is thus a concern that if checks are turned off for actual operation (typically by not producing the corresponding object code), a different program is produced than the one tested. Possible solutions include: using "safer" languages which make it less likely those faults (e.g., memory overwrites) that may be masked by the code of the checks; reducing the effects of disabling the checks (e.g., making their execution conditional on a flag variable, that is reset for operational use, so that disabling the checks does not change the allocation of variables, should avoid the problem of memory overwrites being "unmasked" when disabling the checks); leaving the checks in place in the operational version (a reasonable practice in critical applications, unless performance limitations then make the application unfeasible); and, as a most general precaution, testing the check-less, operational version back-to-back with the version that contains the check: any discrepancy in behaviour is an event worthy of closer examination.

When debuggers or trace analysers are used, two practical problems exist: these tools are often built for interactive use, so that they cannot be used automatically to run batches of large numbers of tests; and they are often intrusive, e.g., because they depend on inserting additional code in the program under test, or on interrupting its operation with frequent "traps". So, again, the program under test behaves differently (with certain tools, it *is* a different program) from the program in normal operation. The solution is in either using non-intrusive monitoring, for instance via a logic analyser (a solution which will only be satisfactory for small programs, where few variables need to be monitored), or, as above, leaving the monitoring software in place during normal operation.

If this last option is chosen, leading to a self-checking capability in the operational software, care must be taken that it does not result in decreased reliability:

- all exceptions raised must be considered in design and handled properly. Proper handling has to consider both the condition that caused an exception to be raised and which reaction to that condition is most appropriate for the system that includes the software under test[21];

- no assertion must be present that may fail on correct but unusual behaviour, or, if they are present, they must be accounted for in exception handler design and in reliability calculations.

This approach - software design with a self-checking capability - encompasses various forms of so-called *software fault tolerance*[Lyu 1995], from using executable assertions with exception handling, to recovery blocks, N-version or self-checking programming, use of robust data structures, audit programs. Some structures typical of fault-tolerant software may also be used for testing only. In particular, *dual programming* has been used, in which a second version of the software under test is built for "back-to-back" testing: the two versions are run together, and discrepancies (in their outputs only or in their internal states as well) are interpreted as possible failures (or internal errors) [Gilb 1974, Geiger 1979].

Common-mode errors reduce the coverage of such methods, so the "testing" version should be built with all precautions that may reduce their likelihood. For instance, the operational version may suffer from timing or space constraints that compel developers to use error-prone design structures or methods; the "testing" version need not be subject to these limitations. If the application lends itself to a conceptually simpler, but inefficient implementation this can be used for the testing version. This has been done using, for the testing version, executable specification,

21 Cf the case of Ariane V, in which a reaction to an unexpected exception caused system failure.

logic or functional languages with an operational version using an efficient but rather unsafe language. In the development approach known as *rapid prototyping*, imperfect and possibly inefficient prototypes are built early on for the purpose of validating requirements. Such prototypes may be useful in a back-to-back testing set-up, although the possible differences between a prototype and the final product may reduce error detection coverage and also cause false alarms that require manual analysis.

8.4. Design of checks for detecting errors and failures

This section describes the options open to the designer of an oracle. The specific design must be chosen via practical considerations: which options are actually viable for the specific software under test; which tools are available and thus what is the cost of the different specific options; what level of coverage can be expected from each specific design.

To judge whether an output or intermediate result is correct, an oracle may use several kinds of knowledge about the intended behaviour of the software under test. Generally speaking, one would like to refer to the specification of the intended function of the software, and code this knowledge into the oracle. However, this is not always feasible: a check thus specified may be too expensive, or too complex for the oracle to be trusted to be reliably built. Furthermore, the specification itself may be missing, or at least not expressed in a form that readily lends itself to be used for checks. This is often the case for complex interactive programs, for programs implementing heuristics, solving vaguely-defined problems or emulating human expert performance.

Ideally, we would like the oracle to test for some property (of the data it observes) which is both necessary and sufficient for the run to be correct (or error-free). So, the oracle would detect all failures (or internal errors as well). However, quite often we find that it is only practical to check for necessary conditions: incorrect executions may still satisfy these conditions, and thus not be rejected.

There is a spectrum of qualities that data values must have, which we could call:

- *correctness*. The ranges of correct values are those that satisfy the specifications (they do not normally change between different uses of the same component);
- *legality*. Illegal values are values that a variable should never assume, under any circumstance. Thus, for instance, a byte-sized variable used for storing a Boolean variable normally has (or ought to have) only two legal values, out of its 256 possible values;
- *reasonableness*. Unreasonable values are those that we know not to arise in operation. For instance, we expect a boat's speed not to exceed some reasonable upper bound, a person's age to be between 0 and 200 years, and so on. Clearly, the limits of reasonableness depend on the available knowledge about the application and the operational environment. If a software component is used in two different systems, the same variable within it may have two different ranges of reasonable values (cf. the Ariane V case).

All correct values are also legal and reasonable, but it may be difficult to show that a certain value is actually correct. However, reasonableness and legality are necessary conditions for correctness, and can thus be used as a basis for checks in an oracle.

We may also decide to look simply for situations that do *not* violate any necessary conditions for correctness, but seem to indicate a high chance of incorrect behaviour. We may thus be able, in testing, to use checks that if used in actual operation would make the software unacceptably untrustworthy.

8.4.1. Specification Checks

These use the definition of "correct result", given by the specification. For instance:

- for a program which has to find the solutions of an equation, the check may consist in substituting the results back into the original equation;

- for a program which is required to find a route from location A to location B, one can verify that starting from A and following the route one actually reaches B;

- for a program which is required to sort a list, one can verify that the output list is actually ordered.

Notice that, in this last example, an output that passes the test may actually be erroneous. For instance, some element of the list may have been lost, or elements may have been added. One can decrease the likelihood of these problems by also checking that the output list has the right number of elements, or that it actually has exactly the same elements as the input list. These examples indicate how error coverage can generally be increased by accepting higher execution costs for the oracle.

Assertions can often be derived from the specifications of small code segments: e.g., loop invariants, predicates on the inputs and outputs of procedures, and so on. With an adequate assertion specification language, the corresponding code can be generated automatically, to be inserted either in the program under test or in the oracle. There has also been research on deriving oracles directly from formal specifications of a program, when the specifications define the function required, rather than the detailed operation of the programme. This is a promising direction.

8.4.2. Comparison checks: back-to-back testing via dual programming and test with previous systems

For some programs, the specifications are not available as simple input-output functions or relations, but as descriptions of the required sequences of operations. In these cases, specification checks may still be obtained via dual programming, as outlined above. If this requires complex code to be produced, it may be difficult to obtain confidence of high error coverage; still, these methods may, for some systems, be the best option available. A variant of dual programming, for software implementing continuous (or mostly continuous) mathematical functions, is *data diversity*: for each selected input case, the software under test is tested twice, first with the selected input case and then with another one, obtained by adding to it a small random error. If the results of the two test runs are very different, there is a suspicion that one of the two may be erroneous. Notice that for the purpose of reliability evaluation only one of the two tests should usually be considered, as the two are not independent samples.

When a reference system is available, like a previous version of the software under test, or a (possibly non-software based) version of the system for which the software is intended, an option is to compare the behaviours of the two versions thus obtained. Again, discrepancies cannot always be ascribed to bugs, as differences between the two versions may be allowed by the specifications.

With back-to-back testing, there may be a problem with spurious alarms. These may be caused, for instance, by:

- the small arithmetic errors that are unavoidable in numeric computations, especially with floating-point arithmetic: the results of two versions are different, but both correct according to the specifications. One can then set thresholds on the allowable discrepancies between the (output or internal) variables that are compared;

- divergence in the behaviour of two versions as an effect of the above phenomenon: a slight numerical difference may affect a variable used as the argument of a decision statement, like an IF statement. then, two correct versions may take different branches of the decision. Manual analysis is often necessary. If this phenomenon is too frequent, the "test" version could be instrumented to be forced to follow the "tested" version's decisions, but obviously this also brings a risk of decreased coverage;

- especially with control programs, small numerical differences may accumulate over time and cause the long-term internal states of the versions to drift apart until discrepant decisions are taken. This is especially likely when using a plant or simulator for test generation, as one of the two versions will then be actively controlling the plant or simulator, while the other will be running "open-loop". The problem may be controlled by forcing on the "testing" version" a state that is consistent with the "tested" version. In practice, this is equivalent to reducing either the coverage of the oracle, or the duration of test runs (each run now spanning the period between two "forcing" operations on the internal state of the testing version), which in turn requires attention on preserving independence between test runs;

- the fact that the specifications allow for completely different, correct outputs for a given input case. For instance, the specifications of a safety interlock function may state that it must guarantee mutual exclusion between the executions of two "requests", but allow the implementation to decide arbitrarily between two requests that are both presented within a short time window. For this kind of situation, manual analysis of each alarm issued by the oracle is usually needed. In some cases, the testers may be able to provide a thorough, automatic correctness check to be invoked only after an alarm.

8.4.3. Timing checks

A special case of specification checks for real-time software consists in monitoring the timing behaviour of the software against its specified timing, looking for executions that last too long, and late (or premature) responses to events. Even when the timing is not precisely specified, designers may have expectations on the duration of specific executions, and an oracle may be instructed to flag unexpectedly long executions as suspicious, for examination by the human testers. Even when no such expectation on "normal" timing exists, it may be formed by observing the behaviour of the software while it is being tested, forming statistics of its execution times for different kinds of test runs, and then flagging test runs that depart from this norm (outlayers) for manual analysis.

In addition to observing the timing of the inputs and outputs, the internal timing behaviour of the software under test can typically be observed via non-intrusive means like logic analysers, by monitoring any time-out detection mechanisms that are built into the system under test, or, when testing on a test host, by instrumenting the code with additional code for measurement. This last function is available in many development environments to allow performance tuning. It is an intrusive method which may allow measurement of many more time variables than other methods, albeit usually with limited accuracy.

8.4.4. Reversal Checks

This is a special case of a specification check. If the software under test is specified to compute a mathematical function, output=F(input), and this function has an inverse function, F', such that F'(F(x))=x, we can compute F'(output) and verify that F'(output)=input. The most elementary case is that in which F is the copy function, e.g. if the software has to write data to disk. The oracle could then read the data back and compare them with the copy in main memory. If F is the square root function, the check would consist of squaring the output and comparing it to the input.

8.4.5. Generic reasonableness checks

There are many software behaviours that are possible but inadvisable, either in any reasonable programming practice or by specific design rules used in a project. These behaviours can be detected and used as error indications. Examples are:

- triggering error-detection traps in the hardware (divide-by-0, illegal opcode, memory boundary violations, write to code memory, etc.) or in the executive or operating system software (illegal parameters to system or library calls);
- assigning out-of-range values to program variables;
- use of uninitialised variables, type violations in use of variables and pointers;
- improper allocation /deallocation of memory (leaks).

8.4.6. Application-specific reasonableness checks

Knowledge about the application domain normally allows one to specify many reasonableness tests on program variables:

- reasonable ranges for variables that represent physical sensor readings (temperature, speed, ..) or database entries;
- acceptable variations between successive values assigned to the same variable, as the rates of change of temperature, speed, acceleration etc. are subject to physical bounds. These bounds may be known with better precision during testing than during actual operation, and even altered under control of the testers;
- known relationships between values of different variables (e.g. as dictated by physical laws);
- acceptable transitions between modes of operation.

8.4.7. Safety Checks

These amount to testing for actions or conditions that are specifically forbidden by the specification, for example for reasons of safety or security. An advantage of these tests is that these constraints are often more simply specified than the operational specifications, making the checks easier to design; and that constraint violations often correspond to especially critical failures, for which better coverage is desired.

8.4.8. Coding and data structure checks

Some applications are explicitly meant to produce data with built-in redundancy, such as codes or redundant data structures (e.g., a file system allowing detection of pointer corruption). Clearly, an oracle may take advantage of these possibilities. A possible use for these forms of information redundancy is to check the redundant information more frequently than would be done in actual operation of the software under test. Audit programs are often used for this purpose, that run independently of the application. These can be run more frequently during testing.

8.4.9. Environment requirement checks

If a simulator of the environment is available, it allows a check that the software satisfies its broad requirements, rather than its detailed specifications. For instance, in testing autopilot software one can verify that the simulated craft stays on its route within the required precision. Such checks may have insufficient coverage for software failures: many failures, although they violate the specifications, may not disturb the behaviour of the controlled environment. Some failures which would disturb that behaviour may also go undetected, due to incompleteness in the system composed of the simulator and software under test. On the other hand, these tests have the advantage of being "end-to-end" checks: they may flag errors in the detailed software specifications with respect to the actual application requirements. A failure probability estimate based on such checks will be a lower bound on the probability of failures of real importance for the controlled environment. Using such checks is thus a form of protection (albeit necessarily limited) against errors in the derivation of detailed specifications and of other checks.

8.5. Other practical indications for automated testing

8.5.1. Costs and compromises

Complete realism in testing may bring high costs as well as other disadvantages. We already discussed the issue of independent sampling and how reasonable approximations to it may be sought.

Other compromises may bring important benefits:

- running the tests on computers that are faster, cheaper or more readily available than the intended target computer (and/or without the full set of other software that should share the same hardware), so as to be able to complete many more test runs. On the down side, this "accelerated" testing may be untrustworthy for any failures that are due to compiler bugs, non-determinism in the hardware (e.g. in processors with caching, superscalar or speculative execution), layout of variables in memory (affecting "overwrite" errors), and other hardware-specific factors.
  Any testing at different speeds than the speed of the target computer is also likely to distort the behaviour of the software under test in terms of timing-dependent failures, overloads, race conditions and such.
  A combination of some accelerated testing and some testing on the actual target computer is often appropriate;

- using test generation methods that only offer limited accuracy in choosing representative samples from the operational profile. For instance, using a simulator that does not model some aspects of the intended environment, or using a trace from runs of a previous system in "open loop" fashion (i.e., ignoring the fact that the outputs of the software under test would in reality affect the sequence of future inputs). The danger is that the test cases chosen will not be representative of the operational profile. Checks for limiting this danger are indicated in the section on "Verifying that a synthetic profile is representative of the intended distributions of key variables";

- if the error in test case generation consists *only* in excluding a set of test cases, the maximum optimistic error that this may cause is equal to the probability of that set being selected in operation. One may thus try to simplify the test case generation problem by excluding specific, rare test cases or subprofiles.

8.5.2. Indications for software design

The practical demands of organising statistical testing will be more or less easily satisfied depending on aspects of how the software under test is constructed. We list a few design features that are useful from this point of view. Most of them also have other advantages in making software more reliable, more predictable or easier to debug and to verify:

- periodical resetting of the software during operation (sometimes called *software rejuvenation* [Huang 1995]). This measure often improves reliability because erroneous internal variables are overwritten before they may cause a failure, memory leaks are less likely to reach catastrophic proportions, etc. For statistical testing, it allows test trajectories with a finite duration (giving better confidence of accurate sampling) and makes it easier to produce independent test runs[22]. For some programs, reset can be a complete re-initialisation. For others, some long-term memory of the results of execution history must be preserved, in persistent variables or files, and all other variables re-initialised from scratch; this organisation also helps in building error recovery and task migration capabilities for fault tolerance;

- time-triggered design (static scheduling with pre-planned interactions among components). This measure is often recommended to make the software's behaviour more predictable. For statistical testing, it gives greater confidence that the execution during testing can be accelerated or slowed down, with respect to real speed on the target computer, without distorting the failure behaviour concerning overloads, race conditions and such. Besides, changes to other software in the completed system are less likely to alter the failure behaviour of the software of interest;

- languages and execution environments that avoid some patterns of hardware- and compiler-dependent failures. For instance, languages that prevent array overflow, mismatch between sizes of memory access and size of the variable stored in the accessed location, pointer manipulation. For statistical testing, they again improve confidence that test can be carried out on a different platform than the target computer and yet give relevant results for predicting operational reliability;

- languages and execution environments that reduce the degree of "non-determinism" in software behaviour. For instance, preventing the use of uninitialised variables, and protecting the address space of the software of interest against accidental changes by other software, eliminates some variables that affect failure behaviour but for which the tester cannot easily specify an operational profile.

[22] Because it avoids dependence due to the state of the internal variables. One still has to verify that no cause of dependence is introduced by the input variables.

9. Analysis of software failure data obtained from testing in order to assess and predict reliability

This section explains how the results of tests can be used to forecast reliability in operation. In practice, a tester will use these methods via a pre-packaged tool, which takes as inputs the observations produced during testing and produces reliability estimates. The purpose of this section is mainly to explain the meanings of the different kinds of estimates, so as to allow an appropriate choice and help in avoiding misuse. An appendix contains more detailed descriptions and explanations of the underlying mathematics.

Note: this section is not relevant for the use of statistical testing for finding and removing bugs.

9.1. Types of reliability estimation problems and inference techniques

Estimating the reliability of software from test data is a problem of *prediction*. The purpose of the estimation is to predict (the probabilities of) different future behaviours, on the basis of the behaviour observed up to the present moment.

This can be done under two regimes of observation:

- considering the results of testing the current version of the software ("as delivered", *steady-state* reliability estimation); the theory underlying this prediction is much the same as used in predicting reliability of physical objects from sample testing;

- considering the series of changes (debugging) which led to the delivered software, and observations from testing each successive version that was thus produced. In this latter case, more evidence can be brought to bear on the prediction. Having observed a trend of (usually) increasing reliability, we can extrapolate this trend into the future. However, any prediction depends on trusting that the trend will continue. In a macroscopic sense, this requires that no qualitative change in the debugging process interrupt the trend (e.g., a change of the debugging team, or the integration of new functionalities could bring about such a change). In a short-term sense, it requires trust that the very last fix to the software was not an "unlucky" one, which decreased reliability.

Prediction is performed by techniques of *statistical inference*. These techniques belong to two different families, which produce information with different meanings:

- the *"classical"* inference techniques are based on the idea of "hypothesis testing". When estimating the value of some measure, these techniques deliver a "confidence interval" with given "confidence level" that the true value is within that interval. This "confidence level" describes how unlikely the experiment performed (e.g., a test campaign on a software product) is to be deceptive. In other words, how unlikely it is that the test results, which support the "hypothesis" that the measure of interest is within the given interval of confidence, would have been obtained *if the hypothesis* were wrong. For instance, the statement "the test campaign has shown that this software has probability of failure on demand lower than 0.001, with confidence 95 %" means "if the probability of failure on demand had been greater than 0.001, then the probability of observing the particular test results that were actually observed would have been lower than 5 %";

- the *"Bayesian"* inference techniques, which are based on the concepts of probabilities as a measurement of uncertainty and how it is reduced by the observation of new evidence. Bayesian techniques produce actual probabilities of the events of interest, and thus avoid decision-making fallacies that would be caused by using "classical" confidence levels as though they were probabilities. A practical difficulty in using Bayesian techniques is that the user must supply *prior* probabilities for the events of interest, i.e., a description of the uncertainty existing before the testing is performed. Whilst there is no definitive prior that will represent ‘complete ignorance’, it is sometimes possible to devise priors that embody conservative beliefs.

In practice, complete algorithms have only been thoroughly studied and published for a few of the techniques mentioned and the possible reliability measures of interest. We will explain methods for:

- steady state estimation of a probability of failure per demand, or of a failure rate, with either Bayesian or classical approaches;

- prediction of reliability growth, using classical inference and ad hoc methods for verifying and improving accuracy.

All the techniques described in this section require somewhat complex mathematical calculations, but these can be automated in software packages. The steady-state evaluation techniques use standard algorithms that are likely to be found as libraries in standard mathematical packages, and in any case easy to program via mathematical software or spreadsheets. The reliability growth modelling techniques have been implemented in a few existing software packages, although none of these does as yet provide the full range of techniques we mention.

An estimate of a reliability measure may take three main forms:

- a point estimate, i.e., a single value for a parameter. For instance, the most obvious point estimate for a Pfd is the "sample mean", i.e., the number of failures observed, divided by the number of test runs. The problem with point estimates is that they indicate that the true reliability of the software under test is "somewhere around" the point estimate, but not how likely the real reliability is to be significantly worse. Usually, this means that an individual estimate may be dangerously far from the true value, although, if one could repeat the test campaign and the estimation many times, the mean of the estimates (and often the mode as well) would be close to the true value[23];

- a lower bound on reliability (a "one-sided confidence interval"), i.e., an estimate with a built-in conservative slant, in the sense that there is a certain degree of "confidence" that the true reliability of the software under test is better than this bound (we explain later the different, more or less straightforward meanings that "confidence" takes in the different estimation methods). The estimate thus includes two numbers, the bound and the level of confidence. One can never have 100% certainty that the true reliability is indeed above this estimated lower bound (unless the bound is 0). A one-sided confidence bound is generally more useful than an unbiased point estimate, in that it takes account of the "measurement error" due to the variability between test campaigns;

[23] For instance, imagine that one computes the sample mean estimator for the Pfd of software that is quite reliable, so that any test campaign has a 50-50 chance of showing no failures. Then, one has a 50-50 chance of producing an estimate of Pfd=0, which is dangerously optimistic. The sample mean has a property that is important in other contexts: it is a so-called "unbiased" estimator, in that, if one were to run the test campaign infinite times, the average of the sample mean would converge to he real value sought (the real Pfd, in this case). Yet, on the single experiment campaign that one performs, the value of an unbiased estimator may well oscillate wildly around this mean.

- a two-sided confidence interval, i.e., a pair of numbers, such that the real value of reliability "can be expected", with a stated "level of confidence" (whose meaning, again, varies between estimation methods), to lie in between the two. These are clearly less useful, for acceptance decisions, than a one-sided confidence bound, although may have some interest as an input for process improvement;

- a direct reliability statement, e.g. a probability of having no failures over a certain period of time. In most cases, this is clearly the most desirable form of prediction.

Point estimates are the least informative, but they may be all that can realistically be obtained for the more mathematically complex estimation problems. In particular, methods for reliability growth prediction, and estimates of system reliability from component reliability, often yield point estimates only.

In the rest of this section, we describe the essential features and differences among the various methods. An appendix contains more detailed explanations and references to the literature.

Throughout this section, we consider reliability estimation concerning just one class of failures. If multiple classes are defined, predictions can be performed separately for each class of failures[24].

9.2. Bayesian vs classical approach

As an example, we consider a software product for which the reliability target has been set as a probability of failure per demand $Pfd \leq 10^{-4}$. The final version of the software is subjected to statistical testing, and passes T test runs without failures.

"Classical" inference procedures answer the question: "how likely would it be for the software to have passed this many tests, if its Pfd were *worse* than 10^{-4}?" They calculate the probability of such a deceptive series of test results occurring, if the software were *less* reliable than required. if this probability is, say, 2 %, we say that there is "98% confidence that the software has $Pfd \leq 10^{-4}$". The basic formula for a required upper bound ϑ_R on the Pfd is:

$$\text{confidence level} = 1-(1-\vartheta_R)^T$$

The confidence level grows with additional successful test runs, so it is an intuitive measure of some aspect of "confidence" in the apparent findings of a test campaign. However, it is unclear how it relates to actual reliability of this specific software. The pair {upper bound on the Pfd, confidence level} is not equivalent to any one statement of how likely the software is to fail in operation. It is common to assume that the confidence bound on a certain measure (in this case, 10^{-4}) *is* the correct value of that measure, but this is incorrect and actually arbitrary, since the confidence bound varies with the chosen confidence level, as is evident from the formula above.

If the "true" requirement for the software is $Pfd \leq 10^{-4}$, which confidence level should the testers be required to reach? If no specific level is required, a supplier could in theory claim any reliability level, after arbitrarily few tests, provided he also declared a correspondingly low confidence level. So, a required confidence level needs to be chosen, but this choice is bound to be arbitrary. Common examples are "apparently high" numbers like 90 % or 95%. However, the confidence level required may become an issue of negotiation between the customer and the supplier. It is recommended that the required level (or, equivalently, the number of test runs required, depending

[24] In a Bayesian treatment, of course, there would tend to be dependence between our beliefs concerning rates of occurrence of different failure types. Observing failures of one type might make us more pessimistic about overall 'quality' (e.g. of the software development process), and thus more pessimistic in our beliefs about the occurrence rates of *other* failure types. In other words, our prior beliefs about the different failures rates would not be independent. See section 9.6 for more details.

on the number of failures observed) is agreed beforehand. As a general indication, the required confidence should be higher when the reliability of the software is more doubtful (a more rigorous criterion is possible with the Bayesian approach).

Another problem arises in decision making: suppose that the requirements for two software components, A and B, are for both to have Pfd$\leq 10^{-4}$. Suppose that A is a small, simple piece of software, produced by a competent team and thoroughly inspected and debugged, while B is a large piece of software produced by a company that is notorious for its poor quality. Which levels of confidence should the tester seek for the two components? Requiring the same level for both components would require the same numbers of successful tests for both. But this is intuitively wrong: if there is evidence that B is less reliable than A, stronger evidence should be required for accepting B than for accepting A. This problem is solved with Bayesian methods.

The Bayesian approach considers that the actual reliability of the software is unknown, so it should be treated as a random variable. In a sense, the program under test has been extracted at random from a population of possible programs. Any one of these programs *could* have been delivered, as far as the observer can tell from the available information (e.g., about the development process used)). These programs have different reliabilities. Reliability estimation consists, roughly speaking, in deciding whether the program under test (the program actually delivered) is, among this population, one of those with a high Pfd or with low Pfd. This population is described by a *prior* probability distribution: for each possible values of the Pfd, a probability is stated that the software under test has that value of Pfd (more precisely, a *probability density function* is specified). This prior distribution must describe the knowledge available before testing: for instance, it can be derived from the spread and distribution of reliability values measured for similar programs developed under similar conditions.

Then, passing a certain number of tests shows that the program is much less likely to be, among the population of possible programs, one with very high Pfd. This decrease in probability is higher for the higher values of Pfd. A *posterior* distribution for the Pfd is thus derived, which takes account of the knowledge derived from statistical testing.

With Bayesian inference, one can thus answer the question "How likely is it that this program has Pfd $\leq 10^{-4}$?" with an actual probability (often called a "Bayesian confidence level"). This can be used in all kinds of reliability calculations. Most importantly, one can, given the probability distribution for the Pfd, calculate the probability that the software will survive a given number of demands without failures, i.e., the probabilities of events of actual interest[25].

A practical obstacle to the adoption of Bayesian methods is that formulating the prior distributions for the variables of interest may require somewhat subtle probabilistic reasoning. Establishing a prior distribution should *not* generally be the responsibility of the tester. The choice of a prior may be a matter of negotiation between the customer and the supplier. If there are widely different perceptions between the parties, the only practical solution may be to adopt the more pessimistic one. The difficulty may be alleviated by checking how sensitive the predictions are to the variation between the different priors that appear plausible (it may be found that the test results "speak for themselves", making the differences in the priors irrelevant). Last, in some prediction problems a prior can be adopted by agreement that can plausibly be classified as representing "ignorance". Examples are given in the Appendix. By giving up the ability to account for evidence prior to testing, the Bayesian approach is thus reduced to share the apparent objectivity of the classical approach, while still being superior in describing a rigorous derivation of a true prediction from stated premises.

[25] Knowing that, say, the program has a 95 % probability of having the required Pfd leaves us uncertain about the "remaining 5%" (the "tail" of the probability distribution for the Pfd). essentially, out of 100 programs that pass my tests, I expect about 95% to be satisfactory, but mine might be one of the other 5%, which could be arbitrarily unreliable.

For these reasons, Bayesian inference procedures should generally be preferred. However, classical methods are often preferred in practice for reasons of familiarity and *apparent* objectivity, and are clearly a legitimate means of agreeing on an acceptance criterion, provided that their results are not misinterpreted. It must also be considered that the two types of methods *often* produce similar reliability estimates, particularly when large amounts of data are available.

Classical inference procedures are easily mechanised, and most formulas that may be useful with statistical testing can be found in reference books and libraries of functions for mathematical software. Bayesian procedures are often computationally more complex, but, for those described here, formulas and pre-fabricated tools are as accessible as for classical methods.

Both approaches are illustrated in more detail in the Appendices, which also contains tables of the basic formulas for steady-state estimation.

9.3. Steady state reliability estimation

We now proceed to summarise the main reliability prediction methods in use. Although reliability growth models have historically received more attention in the software engineering literature, we start with the conceptually simpler problem of steady-state reliability estimation.Steady state reliability estimation, which only considers the test behaviour of the software after the last fix, appears more appropriate than reliability growth based prediction, particularly for safety-critical software, for which it would be dangerous merely to assume that fixes are effective. Although the reliability growth models differ in the details of their assumptions on this point, they all tend to ‘average out’ any short-term reversals of fortune that are caused by bad fixes. In a safety-critical application, however, when we are deciding whether a software-based system is sufficiently reliable as to be fit for purpose, we generally cannot be satisfied with such "on average" results. We would be concerned, for example, with the effect of the last fix, and in some cases would have to assume that this change had produced a new program for which reliability evaluation had to begin anew.

There are two main problems of steady state reliability estimation:

- estimating a probability of failure per demand (Pfd). This is the reference case that we have used in discussing operational profiles. Test cases are selected randomly and independently from the operational profile, and both the number T of test runs and the number n of failures are counted. The essential formulas for reliability estimation are listed in the Appendix.
  With Bayesian inference, an important part of the process is determining the prior distribution. In the Appendix, we explain the general procedure. A special case of "prior ignorance" is also described, which can be used as a baseline prior.

- estimating a failure rate for a continuously-running system. Testing takes place in long stretches of execution, as discussed in Section 7. The mathematical bases of the estimation procedure are similar to those for demand-based systems.

9.4. Reliability growth estimation

9.4.1. Background

It is common for software (or any other system) to undergo a series of design changes ("fixes"), either during development or after release, to remove faults and thus increase reliability. Given the cost of testing for estimating reliability, it is attractive to estimate the reliability that the software will have attained after the last fix, by extrapolating the trend observed as a result of previous fixes.

So, a program is executing in a test (or real) operating environment, and attempts are made to fix faults when these are found as a result of software failures. There is therefore reliability growth, at least in the long term, although there may be temporary reversals when poor fixes cause the introduction of new faults[26]. The reliability growth models attempt to use the data collected here, usually in the form of successive execution times between failures (or, sometimes, numbers of failures in successive fixed time intervals), to estimate the current reliability and predict the future development of the growth in reliability.

Conventionally, this form of prediction is called "reliability growth modelling". Typically, a "reliability growth model" (a prediction method) assumes that the reliability of a program will evolve according to a function of time (and/or of the number of failures observed to date) that belongs to a given parametric family of functions. One measures the failure behaviour of the software as debugging proceeds. For instance, if testing is stopped every time that a failure is observed, and a fix is performed on the software, these measurements consist of the series of times between fixes. Different time variables can be used, as usual, either continuous (e.g., CPU time) or discrete (e.g., number of invocations).

From these measurements, the values of the parameters are estimated for the particular time function describing the evolution of the reliability of the software. Then, computing future values of this function yields the required reliability predictions.

Over the years, many "software reliability growth models" have appeared in the literature (examples are given in the Appendices). Unfortunately, it must be said that no single model has emerged that can be universally recommended to a potential user. In fact, the accuracy of the reliability measures arising from the models tends to vary quite dramatically: some models sometimes give good results, some models are almost universally inaccurate, but no model can be trusted to be accurate at all times. Worse than this, it does not seem possible to identify *a priori* those data sets (program developments) for which a particular model will be appropriate.

Until recently, these problems with the reliability growth techniques have prevented their widespread industrial use. However, in the last few years some powerful new techniques have become available which largely overcome the difficulties. It *is* now possible *in most cases* to obtain *reasonably accurate* reliability measures for software, and *to have reasonable confidence that this is the case* in a particular situation, so long as the reliability levels required are *relatively modest.* The italicised *caveats* here are important, because there are limitations to what can currently be achieved, but they should not be so restrictive as to deter readers from attempting to measure and predict software reliability in industrial contexts.

All these methods involve demanding calculations, and thus the use of software tools. Several reliability prediction tools exist that implement multiple reliability growth models, and a few that support some of the analyses methods described below. See the Appendix for a brief summary of the available tools.

9.4.2. Analysis of predictive accuracy

Since no software reliability growth models has proven to be accurate in general, the best that we can do is to examine whether a certain model is behaving accurately when applied to the evolution of a specific software product under a given operational profile.

It is important to realise that all questions of practical interest involve *prediction.* Thus even if we want to know the *current reliability* at a particular point in this process, we are asking a question about *the future*: in this case about the random variable, *T*, representing the time to the next failure. However we care to express our questions concerning the current reliability - as a rate of occurrence of failures, as a probability of surviving a specified mission time without failure, as a mean time to next failure, or in any other convenient way - we are attempting to predict the future.

[26] It may also be the case that reliability growth tends to an asymptote: as the software improves, the only faults left are so subtle that any attempts at fixing them are likely to introduce new faults that, on average, offset the reliability growth due to successful fixes. This phenomenon has been shown to take place in practice on some large software products, and its existence is accepted, *de facto*, in the practice of many software vendors.

Longer term prediction might involve attempting to estimate the (distribution of) time needed to achieve some target reliability, or the reliability that might be expected to be achieved after a certain duration of further testing.

The important point here is that when we ask, rather informally, whether a model is giving accurate reliability measures, we are really asking whether it is *predicting* accurately. This is something which is sometimes overlooked even in the technical literature: it is sometimes mistakenly claimed that a model is good because a curve of its specific family can be closely fitted, after the fact, to the data series collected until now.

Statistical tests for predictive accuracy essentially ask the question whether a certain method, applied to the early data points in a series, has successfully predicted the later data in the series. The methods that have been more accurate until now can then be plausibly assumed to be also the most accurate for the future behaviour of this data series. This conclusion of course depends on the common assumption of all reliability growth modelling, i.e., that some underlying regularity is present in the debug process. If this assumption is likely to have been violated, e.g. if there has been a change in the testing regime (or in the testing input profile), it should be expected that reliability growth based predictions will exhibit large errors.

Details of these tests are given in the Appendices. Two techniques are described. One, based on *u-plots*, is essentially a test for systematic bias (in the optimistic or in the pessimistic direction) in prediction. The other, based on the *prequential likelihood ratio,* compares the general ability of two models to predict a particular data series, so that one can select the method that has been more accurate over a sequence of predictions. Unlike the *u*-plot, which is specific for a particular type of inaccuracy - a kind of consistent bias - the *PLR* is quite general: the model that it selects as the 'best' is objectively so in a general way. Thus, for example, it can detect those circumstances when the predictions are 'too noisy', and so are individually inaccurate, even when the *u*-plot is good and the predictions show no *consistent* 'bias'.

These tests are not yet widely used, but their use is highly recommended.

9.4.3. Recalibration to improve accuracy of predictions

Recalibration procedures are based on the idea that if a prediction method errs in a consistent, systematic fashion, it should be possible to calculate some kind of systematic correction procedure to compensate for this error. This correction procedure is what is called "recalibration". Recalibration methods have proven effective in practical application, but, again, their use is still limited.

9.5. Reliability estimation for modular software

The previous sections have described different ways in which the reliability of a program can be estimated and predicted from its failure data alone. In these approaches to the software reliability problem no account is taken of the internal structure of the program, which is treated simply as a black box. In practice, however, good design will have resulted in a modular structure for the program, and this can sometimes be exploited in predicting its reliability. (This procedure has been mentioned in section 3.3).

The simplest models that take account of internal structure are based upon a Markov chain to describe the execution of the program. The program has *n* modules, and one and only one of these is executing at any time. It is assumed that each module fails 'purely randomly' in a Poisson process (i.e. the time to failure of the module is exponentially distributed, with a constant failure rate), with failure rate λ_i for module *i*. As the program executes, a succession of modules is executed, with the sequence of the modules determined by a continuous Markov chain.

The detailed failure process is a complicated one, essentially involving switching between different failure processes (modules) with different failure rates (see [Littlewood 1979]). However, in practice the rates at which the switches between modules take place will be orders of magnitude greater than the individual module failure rates (if this were not the case the program would be *very* unreliable). This means that quite simple expressions can be obtained for the overall program reliability in terms of the module reliabilities and parameters of the Markov chain.

For example, assume a program comprises *R* modules, denoted 1, 2, . ., *R*.

The steady-state failure rate of the program will be $\sum_{c=1}^{n} \lambda_c f_c$, where f_c is the limiting proportion of execution time spent in module c (which can be derived from the known values of the transition rates, or estimated experimentally), and λ_c is its failure rate . This result is intuitively plausible, because it says that each module contributes to the overall failure rate of the program in a way that is proportional to both the time spent in the module, and the module's own failure rate. Similar results, of equal simplicity, are available when the operational profile is more generally represented by a semi-Markov process, i.e. when the times spent in individual visits to a module are not exponentially distributed, but have more general distributions.

The usefulness of expressions like these is that they separate information on the operational profile, represented by the parameters of the Markov chain, from the information on the reliabilities of the individual modules. Thus we might be able to compute the reliability of a program in a completely novel operational environment so long as the necessary Markov parameters for the new environment were known [Cheung 1980]. Of course, there is an assumption here that the reliabilities of the modules themselves do not change from one environment to another, merely their frequencies of execution and sojourn times.

If these assumptions are satisfied, at least approximately, these results offer a useful "divide and conquer" approach to the estimation of the reliability of a program in a new operational environment. The information on the Markov parameters can typically be collected very much faster and more cheaply than the information on module reliability, since the latter requires observation of actual failures which may be infrequent. There is thus an important "reuse" of the reliability data from the previous use of the modules. Notice, incidentally, that this module reliability data could be obtained by applying the techniques of previous sections to *module* failure data.

A word of caution is necessary about all the methods that combine "partial" reliability estimates. In this section, as in the case of estimating reliability for "subprofiles" or "subdomains", the reliability of a system is correctly expressed as linear function (a weighted sum) of a set of individual (e.g., module) reliabilities. However, a linear combination of *estimates* of reliabilities does not necessarily produce the corresponding estimator for system reliability. In more detail:

- using unbiased point estimators (like the fraction of test cases that produced a failure) produces an unbiased estimator of system reliability. However, we have pointed out that these estimators are not very convenient for decision-making;

- using one-sided confidence bounds for the modules does not usually produce a confidence bound with the same level of confidence for the system.

The most common solution is simply using the estimates for the components as though they were the true reliabilities, accepting the corresponding loss of accuracy of the predictions. Methods have been studied for building confidence bounds for system reliability from information on component reliabilities. These are somewhat complex. The reader is referred for instance to [Mann 1974, Martz 1982]; caution may be needed regarding the assumptions of these methods, especially that of independence among failures. A simple, but very pessimistic approach may at times be sufficient. Consider a system, with components C_1, C_2, ... C_n, where R_i is the reliability of component C_i, the required system reliability is R_R and system reliability is given by a weighted sum: $R= \sum_{i=1}^{n} a_i R_i$.

If the components have been assigned reliability requirement R_{R1}, R_{R2}, ..R_{Rn} such that:

$$\sum_{i=1}^{n} a_i R_{Ri} = R_R ,$$

and every component is tested to the same level of confidence in the required reliability, i.e., until

$P(R_i \geq R_{Ri}) \geq K$, with K a constant common to all components, it can easily be shown that

$$P(R \geq R_R) \geq P(R_1 > R_{R1}, P_2 > R_{R2}, ..., P_n > R_{Rn}) \geq 1- n(1-K)$$

So, for instance, with 4 components and 99% confidence that each component has the required reliability, the confidence that the system has the required reliability is at least 96%.

Some more indications about this problem are given in the section 9.6.

9.6. Advanced methods for incorporating knowledge about software structure/quality

This section summarily describes some applications of Bayesian inference that have been proposed in the literature in order to use with more accuracy the information available about the software under test. These have only been documented in research papers so far, and their application is likely to require the help of an expert for tailoring them to the specific situation. However, it appear useful to mention their existence, also in view of the increased availability of software tools supporting non-mathematicians in the use of Bayesian methods on complex problems.

The method described in the previous section for estimating the reliability of software taking account of its internal structure raises interesting considerations. Consider a division of the software into modules such that each input case causes only one module to execute. One can then divide the inputs pace into subdomains ("bins") corresponding to the individual modules invoked, perform reliability estimation for each module separately, and combine the results. If one uses an "ignorance" prior for each subdomain, the reliability estimate for the whole software becomes more pessimistic than it would be without binning [Miller 1992]. This happens essentially because one is assuming that a successful test only increases confidence in the future behaviour of the software on the particular subdomain from which the test case was drawn, but not about its behaviour on any other test case.

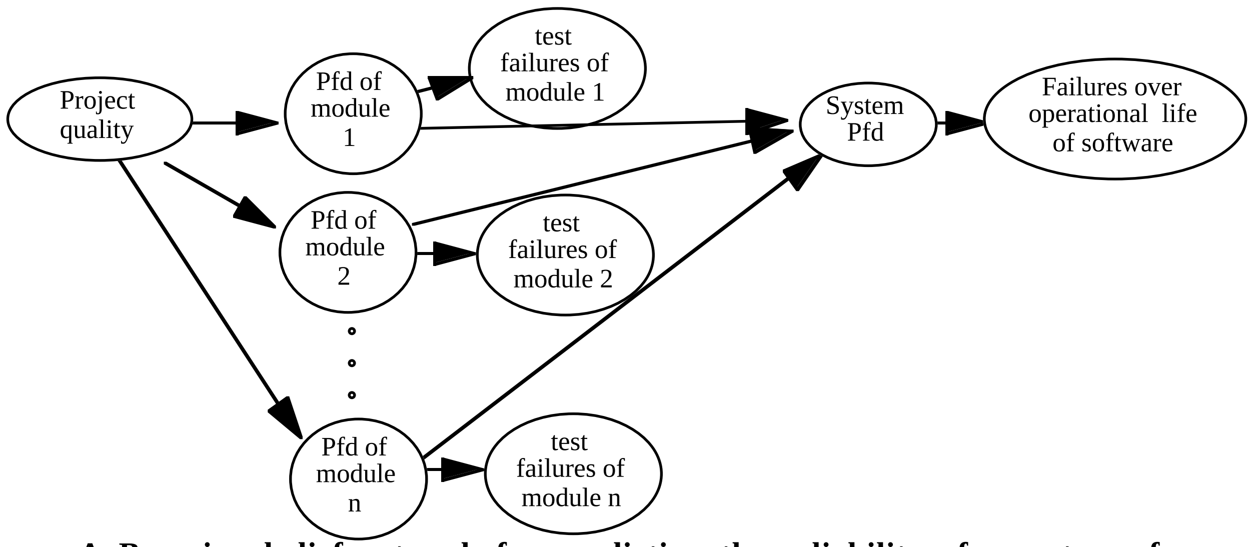
In practice, one would wish to make the subdivision into subdomains, and the priors assigned to them, match what one "really" knows about the software under test. A separate estimation procedure for each subdomain implies that we consider the reliabilities of the subdomains as mutually independent random variables. That is, findings about reliability on one subdomain do not affect expectations about the others. This is usually unrealistic: test cases from any subdomain will cause executions that share at least some code (see [May 1995b] for a discussion of this), and in any case even separate pieces of code are the product of the same production process. For instance, observing very poor reliability on one subdomain increases the probability that the production process "went wrong" in the development of this product, and this ought to be reflected in the prior distributions for the reliabilities on other subdomains (or of other modules), or of the whole software on its whole input space.

As stated earlier, these methods attempt to use more detailed information about the software under test than the methods described earlier. There are two difficulties in their use:

- the likely inaccuracy of the additional information needed. For instance, the method in [May 1995b] assumes that the failure probability per test case is proportional to the amount of code executed in this test case. This is clearly unrealistic, but more realistic assumptions are difficult to establish. This situation can only improve with improved measurement practice in software projects, and recording of reliability data in operation. Predictions about a specific software product can only take advantage of known "statistical laws" observed for similar products. Until a better knowledge is available of the variations between development organisations, this means that such laws should be based on data observed in the same or very similar organisation, for similar software;

- the difficulty of managing the complex mathematical reasoning involved. This problem is being alleviated by the increasing availability of mathematical software. An example follows.

Bayesian belief networks are a method for describing complex interrelationships among diverse information, so as to support Bayesian prediction methods. As an example, consider the following figure.


Project quality
Pfd of module 1
test failures of module 1
System Pfd
Failures over operational life of software
Pfd of module 2
test failures of module 2
Pfd of module n
test failures of module n

A Bayesian belief network for predicting the reliability of a system of modules, taking account of the fact that all modules were the product of a single project and therefore are likely to be of similar quality. This BBN represent reasoning for a known operational profile and a fixed number of test cases per module.

The graph in this figure describes reasoning about a system of modules which are always executed separately, but were developed together. Each node represents an event or random variable, which may be either observed or estimated. The arrows indicate dependence: for instance, knowing the project quality would allow one to refine one's expectations about the Pfd of each individual module. The user must describe in a *node probability table* for each node how likely each possible value is for that node, given a certain combination of values of its "parent" nodes. In this figure, the tables for all the "Pfd of module i" nodes would be filled on the basis of experience. They would represent the fact that the reliabilities of the individual modules may differ over a wide range, but there are project-wide factors that affect the statistical expectations for all of them. The other node probability tables would be filled using the mathematical laws of reliability.

Other examples, and a more extensive introduction to BBNs, can be found in [Delic 1997]. In general, one can use BBNs to describe reasoning that establishes prior distributions (to be updated using statistical test results), taking account of disparate pertinent evidence, like the quality of the development process, the experience of the developers, etc. Clearly, this will only give substantial benefits when sufficient experience has been accumulated for such information to be considered a strong predictor of reliability.

Appendix C. Mathematical explanations and examples

This appendix provides some additional information that may be required in reading the main text, background material about reliability prediction and the reference formulas needed for the basic calculation of steady-state reliability. Further details of the mathematical methods, as applied to software reliability, and appropriate references can be found in [Lyu 1996], Chapters 1 through 4 and Appendix B.

C.1. Sections 1 through 6

C.1.1. Markov chains

Markov chains are a language for representing processes that develop in time. They are widely used to describe reliability and performance evaluation problems. A Markov chain is a graph: a set of *nodes* connected by oriented *edges*. The nodes represent *states* of a system, and the edges represent *transitions* between states. A Markov chain gives a compact representation of a great number of possible behaviours, i.e., sequences of transitions between states. In general, when the system is in a given state, several transitions are possible, one for each edge leaving the node. Each transition is labelled with an indication of probability. If the system modelled has some kind of natural "clock", a *discrete time* Markov chain can be used, i.e., the transition are assumed to happen at clock ticks, and the label on each edge represents the probability of a transition along that edge. Otherwise, a *continuous time* Markov chain will be used, in which each label represents a transition *rate* over time.

By defining a Markov chain, one specifies the probabilities of all different histories of the represented system. Normally, these will depend on the initial state of the system, which must also be defined. There are standard solution methods that then allow one to derive, e.g., the probability of the chain being in a certain state at a given time. In addition, one can obtain *steady-state* probabilities of the various states, i.e. probabilities a long time after the time origin of one observation, which no longer depend on the initial state.

In a Markov chain one can represent how the fact of the system being in a certain state affects the further transitions of that system. However, these are not affected by the history of previous transitions ("limited memory" property). Therefore, there are systems which cannot be conveniently represented by a Markov chain. In addition, the transition rates are a set of constants. This means that the time the system spends in any given state (sojourn time) is exponentially distributed (continuous time) or geometrically distributed (discrete time). A chain thus defined describes a system which evolves through a *Markov process.* A more general category of models (*semi-Markov*) is obtained by giving the times before each transition any general distribution that is only a function of the states before and after the transition. Many reference books are available on Markov chains.

C.1.2. Mathematical definitions of important special cases of stochastic processes

Some special cases of assumptions will often recur about the processes affecting failures:

- in a *stationary* process, the probability of observing any given values for its variables does not vary over time. For instance, the probability of given input values, or of failure, does not vary over time. This implies that sampling, say, a one-minute period will yield observations that are (statistically) similar to any other one-minute period. This is clearly a useful property in our telephone switch example. In addition, stationarity usually simplifies the task of predicting the behaviour of a process over arbitrary lengths of time in the future (even 'stationarity' limited to just some aspects of the process may be of help in this sense). An example of stationarity would be given by a spacecraft for which, during a specified mission phase, the probabilities of the various attitude correction commands, and of the sequences thereof, do not change over time;

- in an *ergodic* process, the various parameters of the process can be estimated from a single realisation of the process: the relative frequencies of all events converge to their corresponding probabilities as the length of the realisation (observation period) becomes large. For instance, in a spacecraft, observing 1,000 seconds in one mission may be equivalent, statistically, to observing 1,000 missions each for one second. The former solution is cheaper and gives better confidence that the internal states of the software follow a realistic trajectory. The main remaining problem is then in deciding whether the single observation is long enough to provide a good sample (observed frequencies tend to the corresponding probabilities as the observation time tends to infinity).

  Notice that not all stationary processes are ergodic: for an instance of a process that is stationary but not ergodic, consider a satellite such that random events in the early phases of a mission (say, failures of some subsystems) will determine the history of commands for the rest of the mission, although this history will not show any statistical variation over time. Sampling instants of just one of the possible mission histories will give very different distributions from sampling the same instant in many of the possible missions.

C.1.3. Representativeness "tests": examples

We give some examples of simple, commonly useful statistical tests. For a detailed treatment of statistical tests the reader is referred to an appropriate reference book, e.g. [Cox 1979].

We limit this discussion to the case of a single random variable, for instance, one of the input variables to the software under test. It is most likely that the null hypothesis in statistical software testing will be a so-called *simple* one, i.e. one in which the supposed distribution from which the observations are sampled, $F(x)$, is completely specified. There are, of course, a very large number of types of departure from this hypothesis that might be of interest because they would result in inaccurate estimates of the software reliability. For example, there will generally be an assumption of independence between successive observations. Here, however, we concentrate solely on whether there is evidence that the observations have a distribution different from $F(x)$.

The *sample distribution function* (normally used as the estimator of $F(x)$) is:

$$\tilde{F}_n(x) = \frac{\text{number of } X_j \le x}{n} = \frac{1}{n}\sum \text{hv}(x - X_j)$$

where *n* is the number of observations (the sample size) and hv(.) is the unit Heaviside function:

$$\text{hv}(z) = \begin{cases} 1 & (z \ge 0) \\ 0 & (z < 0) \end{cases}$$

The Kolmogorov statistic is then measured, defined as the maximum vertical difference between $\tilde{F}(x)$ and $\tilde{F}(x)$:

$$D_n = \sup_x \left|\tilde{F}(x) - F(x)\right|$$

Tables of the levels of statistical significance of this are widely available.

When data naturally take on only a finite number of possible values, or it is convenient to group continuous data into a finite set of intervals, the chi-squared goodness-of-fit statistic can be used.

Here the expected frequencies of observation for all possible values of the random variable (or for all possible intervals) are computed, and compared with the observed values to form the test statistic:

$$\sum \frac{(\text{observed frequency} - \text{expected frequency})^2}{\text{expected frequency}}$$

Again, tables of the significance levels of this are widely available.

In both these tests it is assumed that the null hypothesis is "simple", i.e. that the distribution is completely specified. In some instances (although likely to be rare in the case of statistical testing of software) it may be required to test a more general hypothesis. For example, it may be of interest to ask whether some data are normally distributed, without specifying the mean and variance (μ and σ^2, the parameters that characterise this family of distributions). In this example, we could consider the standardised cumulants of the distribution:

$$\gamma_1 = \frac{\kappa_3}{\kappa_2^{2/3}} = \frac{\mu_3}{\sigma^3}, \qquad \gamma_2 = \frac{\kappa_4}{\kappa_2^2} = \frac{\mu_4}{\sigma^4} - 3$$

where κ_r is the rth cumulant, σ the standard deviation and μ_r the rth moment about the mean. Merely from an inspection of dimensions it is easy to see that these standardised cumulants are not functions of μ and σ^2, the "nuisance" parameters. Thus, letting

$$\tilde{\kappa}_2 = \frac{\sum (X_j - \bar{X})^2}{n}, \quad \tilde{\kappa}_3 = \frac{\sum (X_j - \bar{X})^3}{n}, \quad \tilde{\kappa}_4 = \frac{\sum (X_j - \bar{X})^4}{n} - 3\tilde{\kappa}_2^2$$

we could base a test upon

$$\tilde{\gamma}_1 = \frac{\tilde{\kappa}_3}{\tilde{\kappa}_2^{2/3}}, \quad \tilde{\gamma}_2 = \frac{\tilde{\kappa}_4}{\tilde{\kappa}_2^2}$$

which should be close to zero in the case that the true distribution is normal. See [Cox 1979] or similar books for details of these and other tests.

C.2. Steady state reliability estimation

C.2.1. Tables of reference formulas

C.2.1.1. Probability of failure on demand

Estimating the probability of failure on demand. Bayesian procedure	
The required measure is the probability of failure on demand, ϑ.	
Given T independent test runs with r failures; given a prior distribution with density $f(\vartheta)$:	
<u>General case</u>:	
the posterior density function is:	$f(\vartheta \mid R=r, T) = \dfrac{f(\vartheta) \binom{T}{r} \vartheta^r (1-\vartheta)^{T-r}}{\int_0^1 f(x) \binom{T}{r} (x)^r (1-x)^{T-r}\, dx}$,

and the expected value of Θ is: $E(\Theta | R=r, T)=\int_0^1 f(\vartheta | R=r, T)\, \vartheta\, d\vartheta$

In the special case of *zero failures*: (r=0), $f(\vartheta | R=0, T)=\frac{f(\vartheta)(1-\vartheta)^T}{\int_0^1 f(x)(1-x)^T dx}$

Discrete numerical approximation:

Approximate the continuous prior distribution by assigning non-zero probabilities to a discrete sequence of (increasing) values of Θ:

$$a_i = P(\Theta = \vartheta_i),\ i=0, 1, ..., K, \quad \vartheta_0=0<\vartheta_1<\vartheta_2<...<\vartheta_K=1 \ ;$$

the posterior probabilities are: $P(\Theta = \vartheta_i \mid R=r, T)=\frac{a_i \binom{T}{r} \vartheta_i^r (1-\vartheta_i)^{T-r}}{\sum_{j=0}^{K} a_j \binom{T}{r} \vartheta_j^r (1-\vartheta_j)^{T-r}}$

In the special case of *zero failures* (r=0): $P(\Theta = \vartheta_i \mid R=0, T)=\frac{a_i (1-\vartheta_i)^T}{\sum_{j=0}^{K} a_j (1-\vartheta_j)^T}$

"Beta" prior distribution with parameters (a,b), mean $\frac{a}{a+b}$:

posterior distribution: is "Beta" with parameters (a+r, b+T-r)

expected value of Θ: $E(\Theta | R=r, T))=\frac{a+r}{a+b+T}$

In the special case of *zero failures*: (r=0), $E(\Theta | R=0, T))=\frac{a}{a+b+T}$

"Ignorance" prior, $f(\vartheta) \equiv 1$, for all $\vartheta \in [0,1]$

posterior distribution: is "Beta" with parameters (1+r, 1+T-r)

expected value of Θ: $E(\Theta | R=r, T))=\frac{1+r}{2+T}$

In the special case of *zero failures*: (r=0), $E(\Theta | R=0, T))=\frac{1}{2+T}$

One-sided confidence bounds, Bayesian procedure

General formula: $Pr(\Theta < \vartheta_B \mid R=r, T)=F(\vartheta_B)=\int_0^{\vartheta_B} f(\vartheta | R=r, T)\, d\vartheta$

For the discrete numerical approximation: $Pr(\Theta<\vartheta_B \mid R=r, T)=\sum_{i=0}^{B-1} P(\Theta = \vartheta_i \mid R=r, T)$

"Classical" procedure for estimating the probability of failure on demand.

Given T independent test runs with r failures, we want the probability of failure on demand, ϑ:

the typical, unbiased point *estimator* is the proportion of failed runs: r/T

a *one-sided confidence bound* ϑ_B with confidence *C* is given by solving:

$$\sum_{j=r+1}^{T} \binom{T}{j} \vartheta_B^{j} (1-\vartheta_B)^{T-j} \geq C$$

and in the special case of *zero failures*: (r=0), by solving: $1-(1-\vartheta_B)^{T} \geq C$

Taking account of imperfect oracles

***Bayesian estimation*:**

Given an oracle with coverage O (N.B. If there are indications that the coverage varies with ϑ, a function O(ϑ) can be used instead of the constant O in the following formulas).

<u>*General case*</u>:

the posterior density function is: $$f(\vartheta | R=r, T)=\frac{f(\vartheta) \binom{T}{r} (O\vartheta)^{r} (1-O\vartheta)^{T-r}}{\int_0^1 f(x) \binom{T}{r} (Ox)^{r} (1-Ox)^{T-r}\, dx},$$

<u>*When using the discrete numerical approximation*</u>:

the posterior probabilities are: $$P(\Theta = \vartheta_i \mid R=r, T)=\frac{a_i \binom{T}{r} (O\vartheta_i)^{r} (1-O\vartheta_i)^{T-r}}{\sum_{j=0}^{K} a_j \binom{T}{r} (O\vartheta_j)^{r} (1-O\vartheta_j)^{T-r}}$$

The expected values and confidence bounds can be derived from these formulas.

Classical estimation:

a *one-sided confidence bound* ϑ_B with confidence *C* is given by solving:

$$\sum_{r=j+1}^{T} \binom{T}{r} (O\vartheta_B)^{T} (1-O\vartheta_B)^{T-r} \geq C$$

Predicting probability of failure over system lifetime/ set number of demands

(Bayesian procedure only)

If the posterior density function for Θ is $f(\vartheta)$, the probability of r_f or fewer failures over T_f future demands is: $$\sum_{r=0}^{r_f} \int_0^1 f(\vartheta) \binom{T_f}{r} \vartheta^{r} (1-\vartheta)^{T_f-r}\, d\vartheta$$

In the special case of *zero failures:* (r_f=0): $$\int_0^1 f(\vartheta)\ (1-\vartheta)^{T_f}\, d\vartheta$$

C.2.1.2. Failure rate in continuous time

Estimating the failure rate in continuous time. Bayesian procedure.

The required measure is the failure rate, λ.

Given a test duration t with r failures; given a prior distribution for λ with probability density function *f(* λ);

<u>*General case*</u>:

the posterior density function is: $$f(\lambda| R=r, t)=\frac{f(\lambda)\,\frac{(\lambda t)^r e^{-\lambda t}}{r!}}{\int_0^\infty f(x)\,\frac{(xt)^r e^{-xt}}{r!}\,dx},$$

and the expected value of Λ is: $$E(\Lambda| R=r, t)=\int_0^\infty f(\lambda| R=r, t)\,\lambda\, d\lambda$$

In the special case of *zero failures*: (r=0), $$f(\lambda| R=0, t)=\frac{f(\lambda)\, e^{-\lambda t}}{\int_0^\infty f(x)\, e^{-xt}\, dx}$$

Discrete numerical approximation:
Approximate the continuous prior distribution by assigning non-zero probabilities to a discrete sequence of (increasing) values of Λ:

$$a_i= P(\Lambda= \lambda_i),\ i=0, 1, 2, \ldots K,\quad \lambda_0=0<\lambda_1<\lambda_2<\ldots<\lambda_K$$

the posterior probabilities are: $$P(\Lambda = \lambda_i\ | R=r, t)= \frac{a_i\,\frac{(\lambda_i t)^r e^{-\lambda_i t}}{r!}}{\sum_{j=0}^{K} a_j\,\frac{(\lambda_j t)^r e^{-\lambda_j t}}{r!}}$$

In the special case of *zero failures*: (r=0), $$P(\Lambda = \lambda_i\ | R=0, t)= \frac{a_i e^{-\lambda_i t}}{\sum_{j=0}^{K} a_j e^{-\lambda_j t}}$$

"Gamma" prior distribution with parameters (a,b), mean $\frac{a}{b}$:

posterior distribution: is "Gamma" with parameters (a+r, b+t)

expected value of Λ: $E(\Lambda| R=r, t))= \frac{a+r}{b+t}$

In the special case of *zero failures*: (r=0), $E(\Lambda| R=0, t))= \frac{a}{b+t}$

One-sided confidence bounds, Bayesian procedure

General Formula: $$\Pr(\Lambda< \lambda_B\ | R=r, t)=F(\lambda_B)= \int_0^{\lambda_B} f(\lambda| R=r, t)\, d\lambda$$

For the discrete numerical approximation: $$\Pr(\Lambda< \lambda_B\ | R=r, t)=\sum_{i=0}^{B-1} P(\Lambda = \lambda_i\ | R=r, t)$$

"Classical" procedure for estimating the failure rate

Given test runs for a time t with r failures, we want the failure rate, λ:

the typical, unbiased point *estimator* is : r/t

a *one-sided confidence bound* λ_B with confidence C is given by solving:

$$\sum_{j=r+1}^{K} \frac{(\lambda t)^j e^{-\lambda t}}{j!} \geq C$$

and in the special case of *zero failures*: (r=0), by solving: $1-e^{-\lambda t} \geq C$

Taking account of imperfect oracles

***Bayesian estimation*:**

Given an oracle with coverage O (N.B. If there are indications that the coverage varies with λ, a function O(λ) can be used instead of the constant O in the following formulas).

General case:

the posterior density function is:

$$f(\lambda \mid R=r, t) = \frac{f(O\lambda)\, \frac{(O\lambda t)^r e^{-O\lambda t}}{r!}}{\int_0^{\infty} f(Ox)\, \frac{(Oxt)^r e^{-Oxt}}{r!}\, dx}$$

When using the discrete numerical approximation:

the posterior probabilities are:

$$P(\Lambda = \lambda_i \mid R=r, t) = \frac{a_i \frac{(O\lambda_i t)^r e^{-O\lambda_i t}}{r!}}{\sum_{j=0}^{K} a_j \frac{(O\lambda_j t)^r e^{-O\lambda_j t}}{r!}}$$

The expected values and confidence bounds can be derived from these formulas.

Classical estimation:

a *one-sided confidence bound* λ_B with confidence C is given by solving:

$$\sum_{j=r+1}^{K} \frac{(O\lambda t)^j e^{-O\lambda t}}{j!} \geq C$$

Predicting probability of failure over system lifetime/set period of operation

(Bayesian procedure only)

If the posterior density function for Λ is $f(\lambda)$, the probability of no failures over a duration t_0 of future operation is:

$$\int_0^{\infty} f(\lambda)\, e^{-\lambda t_0}\, d\lambda$$

C.2.2. Demand-based systems: estimating the probability of failure on demand

The inference procedures are based on independence among the test runs. With this assumption, and ϑ the probability of failure on demand (Pfd), the number of failures in T demands (test runs), R, has a Binomial distribution, given ϑ:

$$P(R=r) = \binom{T}{r} \vartheta^r (1-\vartheta)^{T-r}$$

and in particular

$P(R=0) = (1-\vartheta)^T$

Within the Bayesian framework we represent our *a priori* knowledge about the parameter of interest, here ϑ, by a prior distribution. In other words, for any given range of values between 0 and 1, we describe the probability that the actual Pfd of the software under test lies in that range, as we can describe it on the basis of available evidence. Let us call this prior distribution $F(\vartheta)$, with density function $f(\vartheta)$.

C.2.3. An "ignorance" prior distribution

There are advantages in using a prior distribution from the conjugate family[3], which in this case is the Beta(a,b) distribution:

$$f(\vartheta)=\frac{\vartheta^{a-1}(1-\vartheta)^{b-1}}{B(a,b)}$$

where $B(a,b)$ is the Beta function and $a>0$, $b>0$ are chosen by 'you' to represent 'your' belief about ϑ prior to seeing any test results.

In some cases it might be possible to use information about the system and its development process to give numerical values for a and b. Here we shall concentrate on the case where no such information is available, and use the "ignorance prior" with $a=b=1$:

$f(\vartheta) \equiv 1$, for all $\vartheta \in [0,1]$

If the system has executed T test runs with r failures, the posterior distribution of ϑ is Beta($a+r$,$b+T-r$):

$$f(\vartheta \mid r,T,a,b)=\frac{\vartheta^{a+r-1}(1-\vartheta)^{b+T-r-1}}{B(a+r,b+T-r)}$$

which reduces to

$$f(\vartheta \mid r,T,1,1)=\frac{\vartheta^{r}(1-\vartheta)^{T-r}}{B(1+r,1+T-r)}$$

for the "ignorance" prior.

For any particular values of r and T observed during statistical testing, we can use these last two expressions to compute reliability estimates for the software under test. For example, the "best" point estimate of the probability of failure on demand is usually taken to be the posterior mean

$$E(\Theta \mid r, T))=\frac{a+r}{a+b+T}$$

More useful than a simple point estimate, however, is a confidence bound. In Bayesian terms, this is expressed as a pair ϑ_B,α such that:

---

3 The conjugate family has the property that both prior and posterior distribution will be members of the same parametric family of distributions. It represents a kind of homogeneity in the way in which our beliefs are represented, and how they change as we receive extra information.

$\Pr(\Theta < \vartheta_B) \geq 1-\alpha$

For a test campaign containing a given number of test runs, we can compute the maximum number of failures that are admissible for the above inequality to be satisfied; or conversely, for a particular number of failures observed, the minimum total number of test runs over which they should be spread.

The following table illustrates this in the case where the reliability target is expressed as 99% confidence that the true *Pfd* is smaller than 10^{-3}, i.e. when (p_0, α) = (0.001, 0.01), and using the uniform, "ignorance" prior distribution. Thus, 4602 successful test runs need to be executed before it can be claimed that the target has been met. If one failure is seen before the run of 4602 demands is complete, then it must be accompanied by a total of 6634 successfully executed demands.

Number of failures, *r*	Total number of test demands, *T*
0	4602
1	6635
2	8402
3	10041
4	11600
5	13104
6	14566
7	15995
8	17397
9	18778

Table 1: Total number of test runs, *T*, needed to claim that the Pfd ϑ is less than 10^{-3} with 99% probability, if there have been exactly *r* failures in the *T* test runs.

Neither of these ways of expressing reliability - as a point estimate or as a confidence bound for ϑ - is *predictive* of future failure behaviour (as explained in 9.2). Merely being 99% confident that the *Pfd* is smaller than 0.001 is not sufficient for us to be able to say how confident we are that the system will survive, for example, the number of demands that are expected in a year.

C.2.4. General case

Choosing a Beta prior distribution makes for easier calculations, but is not strictly necessary. Modern mathematical software has made the calculations required for Bayesian inference much less forbidding even when complex functions are involved. We describe here the general method for applying Bayesian inference to Pfd estimation, and show a simple numerical approximation that can easily be automated. Most formulas can be found in the boxed summaries at the beginning of this appendix.

Having run T independent test runs, and observed r failures, the posterior density function for the Pfd, Θ, is:

$$f(\vartheta \mid R=r, T) = \frac{f(\vartheta) \binom{T}{r} \vartheta^r (1-\vartheta)^{T-r}}{\int_0^1 f(x) \binom{T}{r} (x)^r (1-x)^{T-r}\, dx}$$

and given this, one can derive the "best" estimate of the Pfd as the mean, or expected value, of Θ, as shown in the table.

To simplify the computation of the integrals, a method is to approximate the continuous distribution $f(\vartheta)$ with a discrete one. We thus represent Θ as a discrete random variable, and describe its distribution via a succession

$$a_i = P(\Theta = \vartheta_i),\ \ i=0, 1, ..., K,\ \vartheta_0=0<\vartheta_1<\vartheta_2<...<\vartheta_K=1\ ;$$

in particular we define $\vartheta_0 = 0$ and $a_0 = P(\Theta = 0)$. Notice that this approximation makes it easy to represent a non-zero probability that the software has 0 Pfd, i.e., is fault-free. This is implausible for complex software, but may well be a reasonable belief for simple and well-developed software.

After observing T tests with r failures, we obtain a discrete posterior distribution:

$$P(\Theta = \vartheta_i \mid R=r, T)=\frac{a_i \binom{T}{r} \vartheta_i^r (1-\vartheta_i)^{T-r}}{\sum_{j=0}^{K} a_i \binom{T}{r} \vartheta_j^r (1-\vartheta_j)^{T-r}}$$

In the case of no failures over T tests (r=0), the posterior distribution is:

$$P(\Theta = \vartheta_i \mid R=0, T)= =\frac{a_i (1-\vartheta_i)^T}{\sum_{j=0}^{K} a_j (1-\vartheta_j)^T}$$

The expressions for confidence bounds are listed in the tables in C.2.1.

C.2.4. Prediction of reliability rather than reliability parameters

The Bayesian approach admits a formal and rigorous theory of *prediction*. We can formulate a proper *reliability* requirement as a pair $(T_o,\alpha)$for which

$$P(\text{no failures in the next } To \text{ demands}) \geq 1-\alpha$$

The *Bayesian predictive distribution* for the number of failures R_f in the next (future) T_f demands, if we have seen r failures in the past T test demands, is

$$P(R_f=r_f \mid r,T,a,b)= \int_0^1 P(R_f=r_f \mid \vartheta)\ f(\vartheta \mid r,T,a,b)\ d\vartheta$$

where $P(R_f=r_f \mid \vartheta)= \binom{T_f}{r_f} \vartheta^{r_f} (1-\vartheta)^{T_f-r_f}$

Now, to find the total number of test demands the system needs to execute, including r failures, in order to pass the test, we put $r_f = 0$ and $T_f = T_o$ in the above expression, and solve for the smallest value of T for which the expression exceeds $1-\alpha$. The following example uses the uniform prior, $a=b=1$, as before.

Number of failures, r	Total number of test demands, T, for (T_O,α)=(46, 0.009895)	Total number of test demands, T, for (T_O,α)=(500, 0.097982)	Total number of test demands, T, for (T_O,α)=(1000, 0.178476)
0	4602	4602	4602
1	9229	9450	9681
2	13855	14298	14766
3	18481	19147	19852
4	23107	23996	24938
5	27734	28845	30024
6	32360	33694	35111
7	36986	38543	40198
8	41612	43392	45285
9	46239	48241	50372

Table 2: Total number of test runs, T, that must be performed in order to claim a probability of failure over T_0 future demands as low as α, if there have been exactly *r* failures during test. Notice in each case how close to linear is the increase in *T* with *r*.

Table 2 gives some examples of this calculation for different (T_O,α): these have been chosen so that the first entry in each column is 4602 as in Table 1, so as to provide a comparison of the two different ways of expressing a reliability goal, and of the impact of this choice of goal upon the amount of testing needed.

Once again, at least 4602 failure-free demands must be seen to claim that the reliability goal (T_O,α)=(46, .009895) has been achieved. If a failure occurs during testing before this number of failure-free demands has been executed, then 9221 failure-free test runs must be observed in addition to the one failed run, and so on.

The difference between the three columns is also a non-intuitive consequence of stating a true reliability requirement. The three columns correspond to three requirements that are equivalent from the point of view that all would be satisfied with 4602 failure-free tests (they correspond to the same 99% confidence upper bound of 0.001 on the Pfd). But we can also see the three columns as representing the needs of customers with different planned lifetimes for the software. A somewhat obvious consequence is that a longer projected lifetime (more demands) implies that the observed test series gives lower confidence that no failure will be observed in the future. In addition, the number of successful test runs needed to compensate for test failures grows with the projected lifetime over which confidence is sought.

C.2.5. Continuously operating systems

In a software system that runs continuously, it is usually reasonable to assume that failures will occur "purely randomly", i.e. in a simple Poisson process of rate λ. The theoretical development follows closely that of the discrete case above. The number of failures, *R*, in time *t* has a Poisson distribution:

$$P(R=r)=\frac{(\lambda t)^r e^{-\lambda t}}{r!}$$

and in particular

$$P(R=0)=e^{-\lambda t}$$

The conjugate family here is the Gamma. Thus if we represent our *a priori* belief about the failure rate λ by Gamma(*a*, *b*), the posterior for λ after seeing *r* failures during time *t* is Gamma(*a*+*r*, *b*+*t*):

$$f(\lambda \mid r,t;a,b) = \frac{(b+t)^{a+r}\lambda^{a+r-1}e^{-(b+t)\lambda}}{\Gamma(a+r)}$$

As usual in the Bayesian framework, it is best to use a proper prior distribution that truly represents 'your' beliefs, and there is no obviously plausible "ignorance" prior. In what follows, for illustrative purposes only, we have used the improper[4] uniform prior distribution:

$$f(\lambda) = 1$$

which gives the (proper) posterior distribution for λ:

$$f(\lambda \mid r,t) = \text{Gamma}(r+1,t)$$

For any particular values of *r* and *t* obtained from testing the software, we can use expressions like these last two to compute reliability estimates. Thus the best point estimate of the failure rate is the posterior mean:

$$E(\Lambda \mid r,\, t;\, a,b)) = \frac{a+r}{b+t}$$

As before, a confidence bound is generally more useful than just a point estimate. We can express the reliability requirement as a pair (λ_0,α) such that

$$P(\lambda < \lambda_0 | r,t;a,b) = 1-\alpha$$

The following table illustrates this calculation in the case where the reliability target is 99% confidence that the true failure rate is less than 10^{-3}, i.e. when (λ_0,α)=(0.001, 0.01) and the uniform prior is used. This table is used similarly to the earlier one: thus if precisely one failure has been seen in the test, then the reliability target is reached if the total time on test reaches 6638.35 time units.

Number of failures, *r*	Total elapsed time on test, *t*
0	4605.17
1	6638.35
2	8405.95
3	10045.12
4	11604.63
5	13108.48
6	14570.62
7	15999.96
8	17402.65
9	18783.12

Table 3: Total elapsed time on test, *t*, needed, if there have been exactly *r* test failures, so as to claim 99% probability that $\lambda \leq 0.001$.

Once again, neither of these ways of expressing the reliability (as a point estimate or as a confidence bound) allows predictions to be made of future failure behaviour. We can formulate a *reliability* requirement as a pair (t_0,α) such that

[4] This probability distribution is "improper" in the sense that its integral over the whole range of the random variable is not 1.

$P(\text{no failures in next } t_0) = 1 - \alpha$

Now $P(\text{no failure in next } t_0 | j \text{ failures in } t)$

$$= \int_0^\infty e^{-\lambda t_0} p(\lambda \mid j,t) d\lambda = \int_0^\infty e^{-\lambda t_0} Gamma(j+1,t) d\lambda = \left(\frac{t}{t+t_0} \right)^{j+1}$$

if we use the same uniform prior as previously. Table 4 gives some examples of the calculation for different (t_0,α).

Number of failures, *r*	Total elapsed time, *t*, for (t_0,α)=(46.517,0.01)	Total elapsed time, *t*, for (t_0,α)=(500,0.097940)	Total elapsed time, *t*, for (t_0,α)=(1000,0.178407)
0	4605.17	4605.17	4605.17
1	9233.57	9453.89	9685.78
2	13861.96	14304.05	14771.85
3	18490.36	19154.56	19859.28
4	23118.76	24005.22	24947.26
5	27747.16	28855.95	30035.51
6	32375.57	33706.72	35123.91
7	37003.97	38557.52	40212.41
8	41632.37	43408.33	45300.98
9	46260.77	48259.15	50389.60

Table 4: Total elapsed time on test, *t*, needed if there have been exactly *r* failed demands, so as to claim a probability α of no failures over a future duration of execution t_0. As in Table 2, notice in each case how close to linear is the increase in *t* with *r*.

C.2.6. Discussion

The tables here illustrate some important differences between the different ways of expressing a reliability goal, and they show that the more usual approach, involving a confidence bound for the *Pfd* or the *failure rate*, could be misleading. One danger with this formulation of a reliability requirement is it might be taken to mean that when the test is passed we can, in the *Pfd* example, say, treat the *Pfd* as *actually being* 10^{-3} for all intents and purposes, since the 99% confidence that it is not larger than this is the same as 'almost certainty'. This would be a dangerously misleading view. The 99% Bayesian confidence statement that ϑ is smaller than 0.001 means that there is a 1% chance that it lies somewhere in the interval (0.001,1). However, this analysis does not allow us to take any account of the contribution to unreliability arising from this component of uncertainty.

The advantage of the prediction-based reliability formulations are that they allow us to compute directly the things that are of practical interest, which are *always* predictions for future system behaviour. A comparison of the tables also shows us how much more 'conservative' they are than the confidence bound approach. The examples have been chosen to have the same entries on the first line. That is, in the *Pfd* example, the person with the confidence bound reliability goal would be satisfied that his/her goal had been reached if 4602 failure-free demands were executed, as would the person(s) with the three different prediction-based reliability goals. Thus there is a sense in which these can be regarded as 'similar' reliability goals at the outset, although differently expressed.

What is notable is the way in which, in Tables 2 and 4, the numbers of failure-free demands needed for the reliability goal to be achieved, in the presence of failures, increase much faster than in Tables 1 and 3. That is, expressing the reliability goal as a prediction, rather than as a confidence bound on the *Pfd*, is in a sense a more conservative approach. This alone would seem sufficient reason for mandating the use of the prediction approach, particularly for critical systems.

We have concentrated upon the Bayesian approach here, rather than the more classical "frequentist" one, precisely because it allows a proper formal treatment of prediction. In fact, when we use the uniform priors as in our examples, the frequentist results for *confidence bounds* are very similar to the Bayesian ones**.** In this sense, this choice of prior can be thought of as the one that a frequentist would use if he/she had no prior information.

C.2.7. Classical confidence bounds

We can compute the classical frequentist equivalent of Table 1 as follows: having observed r failures in T test runs, the corresponding entry in the table is the smallest value of N satisfying

$$\sum_{j=r+1}^{T} \binom{N}{j} \vartheta_B^{j} (1-\vartheta_B)^{T-j} \geq 1-\alpha$$

Equivalently, if r failures have been seen in T trials, then for a particular α the largest value of p_0 satisfying the above gives a (p_0, α) confidence bound:

$$\Pr(\vartheta<\vartheta_B) \geq 1-\alpha$$

Note that the interpretation of this is different from that of the Bayesian bound and less intuitively appealing: here ϑ_B is the random variable about which the probability claim is being made, whereas in the Bayesian case the probability statement refers to the unknown ϑ. That is, a Bayesian bound concerns our confidence in (i.e. a probability statement about) the parameter of interest.

It is easy to show that the Bayesian and frequentist results are very similar when the uniform prior is used for the former. In fact entries in Table 1 are smaller than the frequentist ones by exactly 1. In the continuous time case, when the parameter of interest is the rate λ, the frequentist results are identical to those obtained from the Bayesian argument *using a uniform prior* (see Table 3).

C.3. Inference with reliability growth

C.3.1. Examples of reliability growth models

In this short document there is not sufficient space to describe in detail the very many different models that have been proposed to describe the growth in reliability of software. Instead, we shall give just two illustrative examples of well-known models.

The first of these, the Jelinski-Moranda (JM) model, is one of the earliest models that were devised specifically for software [Jelinski 1972]. In this model it is assumed that the successive times to failure of the program are exponentially distributed, that the program starts life with an unknown number, N, of faults, and that each fault contributes the same (unknown) amount ϕ to the overall failure rate. It is also assumed that, following each failure, the fault that caused that failure is removed with certainty. At stage i, then, the time between the (i-1)th and ith failures, T_i, will have an exponential distribution with rate $(N-i+1)\phi$, since i-1 of the original N faults will by then have been removed. In order to estimate this distribution, we need to estimate the two unknown parameters, N and ϕ. This would usually be done by finding the values of N and ϕ that maximise the likelihood function:

$$\prod_{j=1}^{j=i-1}(N-j+1)\phi e^{(N-j+1)\phi t_j}$$

and substituting these into the exponential distribution for T_i. From this we could then express the reliability in any appropriate way: as an estimated probability of surviving for a certain time without failure, as a failure rate, as a mean time to failure, etc.[5]

In the JM model, the growth in reliability is essentially deterministic, since the improvement in the failure rate at each fix is the same amount, ϕ. In contrast, the Littlewood-Verrall model [Littlewood 1973] treats the effect of a fix as itself uncertain. Here it is assumed that T_i has an exponential distribution *conditionally* with a failure rate Λ_i. In order to capture the inherent uncertainty in the attempts to fix faults, Λ_i is assumed to have a gamma distribution, Gamm($\alpha,\psi(i)$), where $\psi(i)$ is an increasing function of i. The idea here is that the successive failure rates will *tend* to increase (technically, they are *stochastically increasing*), but that there is no guarantee that a specific fix will improve the failure rate and it may even make it worse. In most applications of this model, $\psi(i)$ is taken to be the linear function $\beta_1+\beta_2 i$, and a prediction of T_i will require the maximum likelihood estimation of the three parameters α, β_1, and β_2 as for the JM model.

These two examples illustrate the way in which most of the reliability growth models can be used to make predictions. They are parametric models, i.e. they are characterised by the parameters N and ϕ in the case of JM and α, β_1, and β_2 in the case of LV. At any particular stage, i, we use the failure data that have been seen up to that stage, $t_1, t_2, \ldots t_{i-1}$, to estimate the unknown model parameters, and substitute these into appropriate mathematical expressions to estimate the distribution of T_i, or to make other predictions. As more testing data become available, i.e. i increases, we might expect the estimates of the parameters of the model, and so the predictions, to improve in accuracy.

There are many more reliability growth models in the literature. Unfortunately, as we have stated earlier, there is no single one that can be universally recommended so that a user can be confident that the predictions of reliability will be accurate. Instead, we need to apply several models and somehow select the one (or more) that is giving accurate results *for the data set under consideration*. Techniques that allow this analysis to be carried out have recently become available, and we describe these now.

C.3.2. Analysis of predictive accuracy

Consider the simplest prediction problem of all: that of estimating the current reliability. Let us assume that we have observed the successive inter-failure times $t_1, t_2, \ldots, t_{i-1}$, and we want to predict the next time to failure T_i. We shall do this by using one of the models to obtain an estimate, $\tilde{F}_i(t)$, of the true (but unknown) distribution function $F_i(t) \equiv P(T_i < t)$. Notice that, if we knew the true distribution function, we could calculate any of the measures of current reliability that may be appropriate for a particular application.

We now start the program running again, and wait until it next fails; this allows us to observe a realisation t_i of the random variable T_i. We shall repeat this operation of *prediction* and *observation* for some range of values of i. We would say, informally, that the model was giving good results if what we *observed* tended to be in close agreement with what we had earlier *predicted*. We now describe formal ways of comparing prediction with observation.

5 In this discussion, we refer to continuous-time models. Models also exist that use a discrete time scale; the principles discussed in this section apply equally to them.

Of course, our problem would be easier if we could observe the true $F_i(t)$ so as to compare it with the prediction, $\tilde{F}_i(t)$. Since this is not available, we must somehow use the t_i, which is all we have. Clearly this is not a simple problem, and it is compounded by its being non-stationary: we are interested in the accuracy of a sequence of *different* distributions, for each of which we see only one observation. However it is possible to think of simple comparisons we can make. Let us imagine, for example, that we are only interested in having an accurate estimate of the *median*, m_i, of T_i. Remember that the median is the value of T_i that is exceeded with probability $^1/_2$. We could count what proportion of the actual t_i exceeded their predicted medians, $\tilde{m}_i$ and if this proportion were very different from $^1/_2$ we would conclude that the median predictions were poor.

Such an analysis does not tell us a great deal. Even if a series of predictions passed this test, we would only acquire confidence in the medians. It would not tell us whether other measures, such as the mean time to failure or the failure rate function, were accurate. What we really need is to be able to detect *any* kind of departure between prediction, $\tilde{F}_i(t)$, and truth, $F_i(t)$.

Our first general technique for detecting systematic differences between predicted and observed failure behaviour is called the *u-plot* and it is based on a generalisation of the simple median check described above. Details of this are given in the boxed-in text "the u-plot". The *u*-plot can be thought of as detecting a *systematic* difference between the predictions and the truth. This is very similar to the notion of *bias* in statistics: there we use the data to calculate an *estimator* of a population parameter, and this estimator is called unbiased if its average value is equal to the (unknown) parameter. Of course, our case is more complex since at each stage we wish to estimate a *function*, rather than merely a number, and we can only detect prediction error over a *sequence* of *different* predictions because of the inherent non-stationariety of the problem.

If for a particular data set a model gives predictions which have a good *u*-plot, this does not guarantee that they are accurate in all possible ways. On the other hand, a "bad" *u*-plot does indicate a systematic prediction error. Returning to the analogy of estimating a population parameter in statistics, even if we have an estimator that is unbiased we may still prefer on other grounds to use a biased one. For example, the unbiased estimator may have a large variance, so that although its expected value is equal to the unknown parameter, any particular calculated value of the estimator may be very far from this. This is the difference between what happens *on average* and what happens at *a particular instance*. Similar arguments apply to a good *u*-plot, which also tells us about average behaviour, but which can also mask large inaccuracies on particular predictions.

Such considerations bring us to our next technique for analysing predictive accuracy, the *prequential likelihood function*, and the prequential likelihood ratio (*PLR*): see the "PLR" box for details. The *PLR* allows us to compare the ability of two models to predict a particular data source, and select the one that has been most accurate over a sequence of predictions. Unlike the *u*-plot, which is specific for a particular type of inaccuracy - a kind of consistent bias - the *PLR* is quite general [Dawid 1984]: the model that it selects as the 'best' is objectively so in a general way. Thus, for example, it can detect circumstances when the predictions are 'too noisy', and so are individually inaccurate, even when the *u*-plot is good and the predictions show no consistent 'bias'.

It must be admitted that these ways of examining the accuracy of predictions are non-trivial, and users may find them at first quite unfamiliar. However, the *use* of the techniques is really quite straightforward, involving nothing more than simple graphical analysis as we shall see in the following examples.

The u-plot

The purpose of the *u*-plot is to determine whether the predictions, i.e., the distributions $\tilde{F}_i(t)$ of the times to failure after each fix, are 'on average' close to the true distributions, $F_i(t)$. It can be shown that, if the random variable T_i truly had the distribution $\tilde{F}_i(t)$, in other words if the predicted and the true distributions were *identical*, then the random variable $U_i = \tilde{F}_i(T_i)$ will be uniformly distributed on (0,1). This is called the *probability integral transform* in statistics [DeGroot 1986]. If we were to observe the realisation t_i of T_i, and calculate $u_i = \tilde{F}_i(t_i)$, the number u_i will be a realisation of a uniform random variable. When we do this for a sequence of predictions, we get a sequence $\{u_i\}$ which should look like a random sample from a uniform distribution. Any departure from such 'uniformity' will indicate some kind of deviation between the sequence of predictions, $\{ \tilde{F}_i(t) \}$, and the truth $\{F_i(t)\}$.

We look for departure from uniformity by plotting the *sample distribution function* of the $\{u_i\}$ sequence. This is a step function constructed as follows: on the interval (0,1) on the *x* axis, place the points $u_1, u_2, . . u_n$ (each of these is a number between 0 and 1), then from left to right plot an increasing step function, with each step of height $^1/_{(n+1)}$ at each *u* on the abscissa. The range of the resulting monotonically increasing function is (0,1), and we call it the *u*-plot.

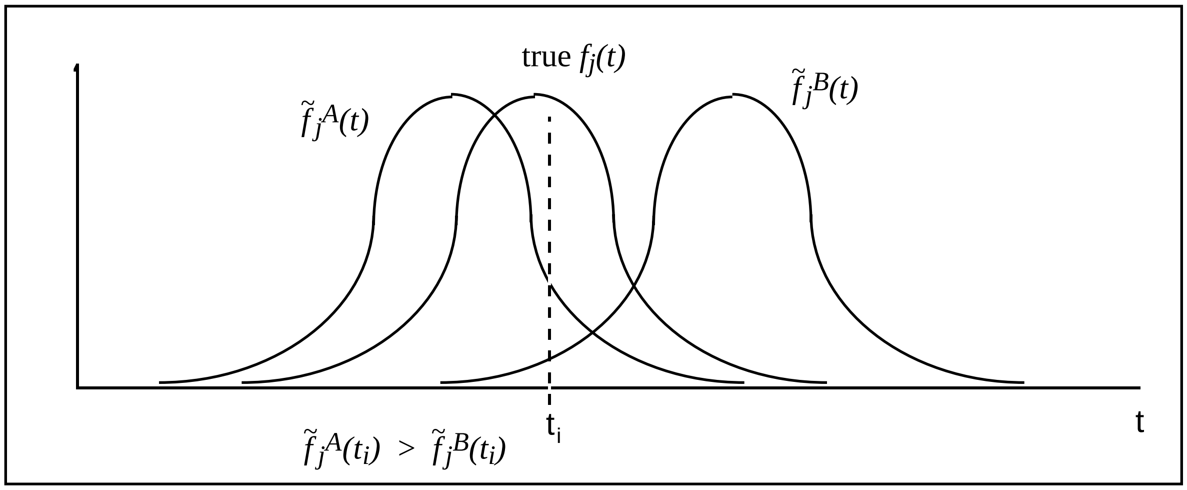
If the $\{u_i\}$ sequence were truly uniform, this plot should be close to the line on unit slope. Any serious departure of the plot from this line is indicative of non-uniformity, and thus of a certain type of inaccuracy in the predictions. A common way of testing whether the departure is significant is via the Kolmogorov-Smirnov (KS) distance, which is the maximum vertical deviation between the plot and the line [DeGroot 1986], for which there are widely available tables. However, a formal test is often unnecessary: it is often clear merely from an informal perusal of the plots that the predictions are poor.

More importantly, such informal inspections of *u*-plots can tell us quite a lot about the *nature* of the prediction errors. For example, if the predictions were consistently too *optimistic* we would be underestimating the chance of the next failure occurring before *t* (for all *t*). The number u_i, in particular, is the estimate we would have made, before the event, of the probability that the next failure will occur before t_i, the time when it *actually does* eventually occur. This number would therefore tend to be smaller than it should be in the case of consistently too optimistic predictions. That means the *u*s will tend to bunch too far to the left in the (0,1) interval, and the resulting *u*-plot will tend to be *above* the line of unit slope. A similar argument shows that a *u*-plot which is entirely below the line of unit slope indicates that the predictions are too pessimistic. More complex *u*-plot shapes can sometimes be interpreted in terms of the nature of the inaccuracy of prediction that they represent.

It is this observation that suggested the "recalibration" procedure described in this document. Sometimes there is approximate stationariety in the errors made by a prediction procedure - the JM model, for example, is usually consistently too optimistic in the predictions it makes; the LV model is often consistently too pessimistic. In such cases, it seems reasonable to use the *u*-plot to estimate this error, and then use this estimate to adjust a future prediction.

The *PRL* - Prequential likelihood ratio

The *PLR* is a means of deciding which of a pair of prediction systems is giving the more accurate results on a particular data source. Consider the figure below,



where there are shown two ways, *A* and *B*, of making a prediction at stage *j*. Here we see the true distribution (in fact the probability density function, *pdf*) of the next time to failure, T_j, together with estimates of this (i.e. predictions) coming from two different models, *A* and *B*. Clearly here *A* is better than *B*. After making these predictions, we wait and eventually see the failure occur after a time t_j. Obviously, we would expect t_j to lie in the main body of the true distribution, as it does here: i.e. it is more likely to occur where $f_j(t)$ is larger. If we evaluate the two predictive *pdf*s at this value of t, there will be a tendency for $\tilde{f}_j^A(t_j)$ to be larger than $\tilde{f}_j^B(t_j)$. This is because the *A pdf* tends to have more large values close to the large values of the true distribution than does the *B pdf*: this is what we mean when we say "the *A* predictions are closer to the truth than the *B* predictions".

Thus if the predictions from *A* are more accurate than those from *B*, the ratio

$$\frac{\tilde{f}_j^A(t_j)}{\tilde{f}_j^B(t_j)}$$

will tend to be larger than 1.

The *PLR* is merely a running product of such terms over many successive predictions:

$$\text{PLR} = \prod_{j=k}^{j=i} \frac{\tilde{f}_j^A(t_j)}{\tilde{f}_j^B(t_j)}$$

and this should tend to increase with *i* if the *A* predictions are better than the *B* predictions. Conversely, superiority of *B* over *A* will be indicated if this product shows a consistent decrease.

Of course, even if *A* is performing consistently more accurately than *B*, we cannot guarantee that $\tilde{f}_j^A(t_j)/\tilde{f}_j^B(t_j)$ will always be greater than one. Thus typically in a case where *A* is better than *B*, we would expect the plot of *PLR* (or more usually, for convenience, the *log* of this) to exhibit overall increase but with some local random fluctuations.

We are usually interested in comparing the accuracy of more than two sequences of predictions. To do this we select one, quite arbitrarily, as a reference and conduct pairwise comparisons of all others against this, as above.

PLR is a completely general procedure for identifying the better of a pair of sequences of predictions. Apart from the intuitive plausibility of *PLR* as a means of selecting between many competing prediction methods on a particular data source, support for this technique comes from a more formal asymptotic theory [Dawid 1984].

C.3.3. Recalibration to improve accuracy of predictions

The basic idea here is to exploit the fact that it is sometimes the case that the prediction errors are approximately stationary. More formally, $F_i(t)$ is approximately equal to $G[\tilde{F}_i(t)]$, for some unknown 'error function' G for all i. The point is that there is always an *unknown* function that will transform the predicted distribution into the true distribution, but it is only sometimes the case that this function is approximately the same for all i. When this occurs, we have the opportunity of *estimating* this error function from the earlier predictions we have made. In fact, it can be shown that the u-plot based upon these earlier predictions is a suitable estimator of G [Brocklehurst 1990]. The recalibration procedure is therefore as follows

1 Obtain u-plot, say G^*, based upon the raw predictions that have been made before stage i[6].

2 Obtain $\tilde{F}_i(t)$ from 'raw model' for prediction at stage i.

3 Calculate recalibrated prediction $\tilde{F}_i^*(t) \equiv G^*[\tilde{F}_i(t)]$.

4 Repeat at each stage i.

The most important point to note about this procedure is that it is truly predictive, inasmuch as only the past is used to predict the future. This means that it is not necessary to believe *a priori* that the recalibrated predictions will be better than the raw ones, since the various techniques for comparing and analysing predictive accuracy can be used. In particular, the *PLR* will tell us whether recalibration has produced better results than a simple use of the raw model.

C.3.4. Example

For this example we shall use the data shown in section 5.2. The software reliability growth models we shall use, with references where details of these may be found, are as follows: Jelinski-Moranda (JM) [Jelinski 1972], Littlewood (LM) [Littlewood 1981], Littlewood-Verrall (LV) [Littlewood 1973], Goel Okumoto (GO) [Goel 1979], Musa Okumoto (MO) [Musa 1984], Duane (DU) [Crow 1977], Littlewood non-homogeneous Poisson process (LNHPP) [Miller 1986] and Keiller Littlewood (KL) [Keiller 1983].

In the example we shall, for simplicity, concentrate upon simple one-step-ahead predictions: at stage i in the testing, when we have seen the first (i-1) failures, we shall use the corresponding inter-failure times as the raw data to predict, from each model, the distribution of the *next* time to failure. In Figure i we show the medians of the distributions calculated in this way (the numbers on the y axis represent seconds of execution time).

6 For technical reasons, which do not detract from the general explanation given here, it is desirable for G^* to be a *smoothed* version of the joined-up step-function u-plot; a spline-smoothed version has been used in the following examples.

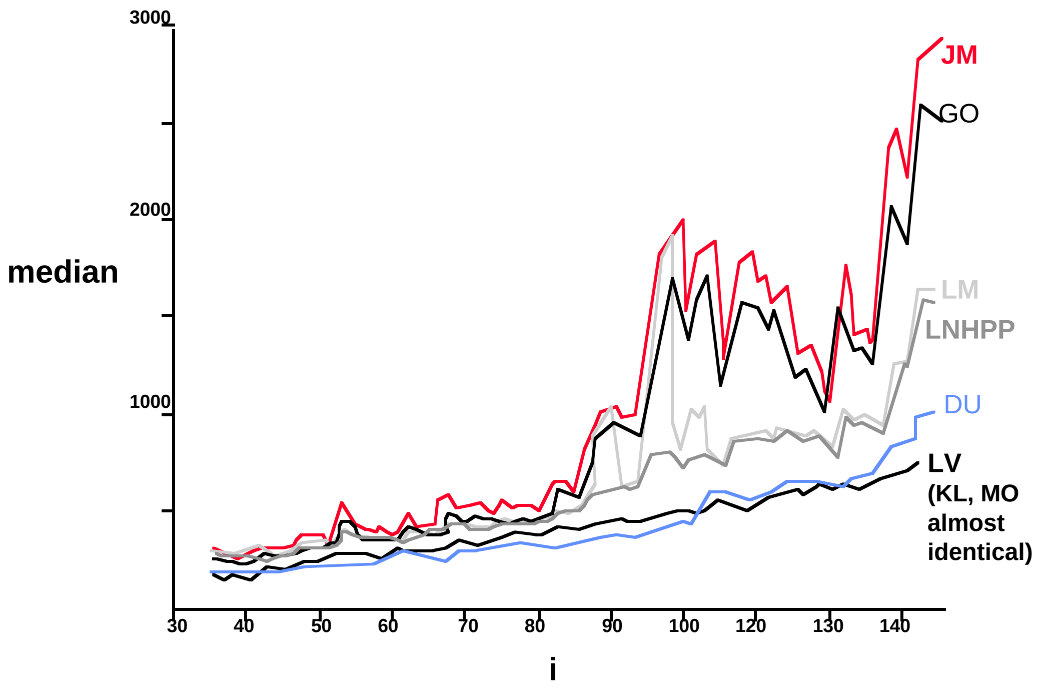


Figure i

In the early predictions of this plot there is reasonable agreement between the eight different sets of median predictions, but from about *i*=85 the models start to diverge in their estimates of the evolving median times to failure. Whilst all model agree that there is reliability growth, some models (JM, GO) are giving much larger estimates (i.e. are more 'optimistic') than others (LV, KL, MO). In addition, some models are far more noisy in their sequence of predictions than others: JM, GO exhibit quite large fluctuations up and down, in contrast to the smoother growth shown by LV, KL, MO.

Whilst the disagreement between the models is clear, there is no information in this plot that allows us to determine whether the JM and GO predictions are *objectively* optimistic, nor whether they are unduly noisy (there could be real noise in the true reliability as a result of bad fixes). To answer questions like this we need to study the u-plots and PLR.

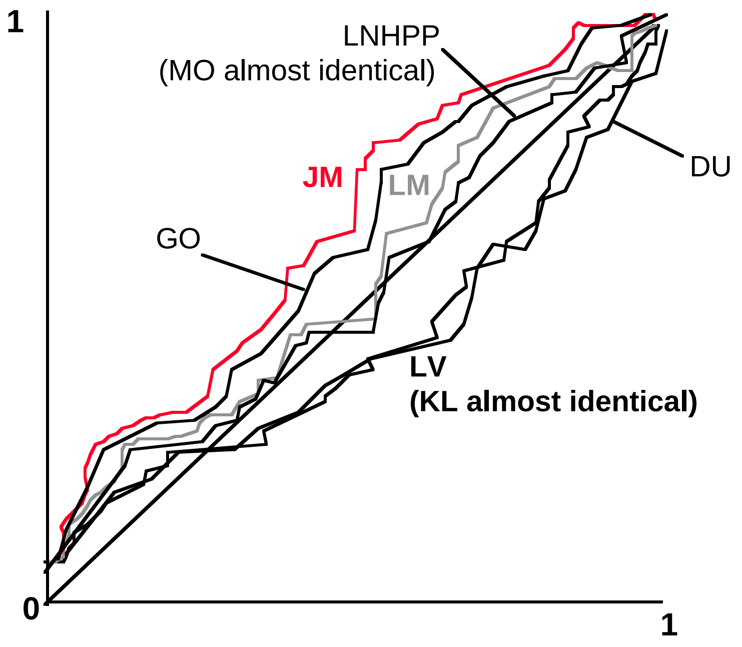


Figure ii: u-plots for the same data

Figure ii shows the u-plots of the different models for these 100 predictions. These show that the JM and GO plots are entirely above the line of unit slope, thus confirming that these predictions are in fact objectively too optimistic. Similarly the KL, LV plots are almost entirely below the line of unit slope, showing that they are too pessimistic. The other models are close to the line of unit slope, indicating that they seem to be accurate according to this test.

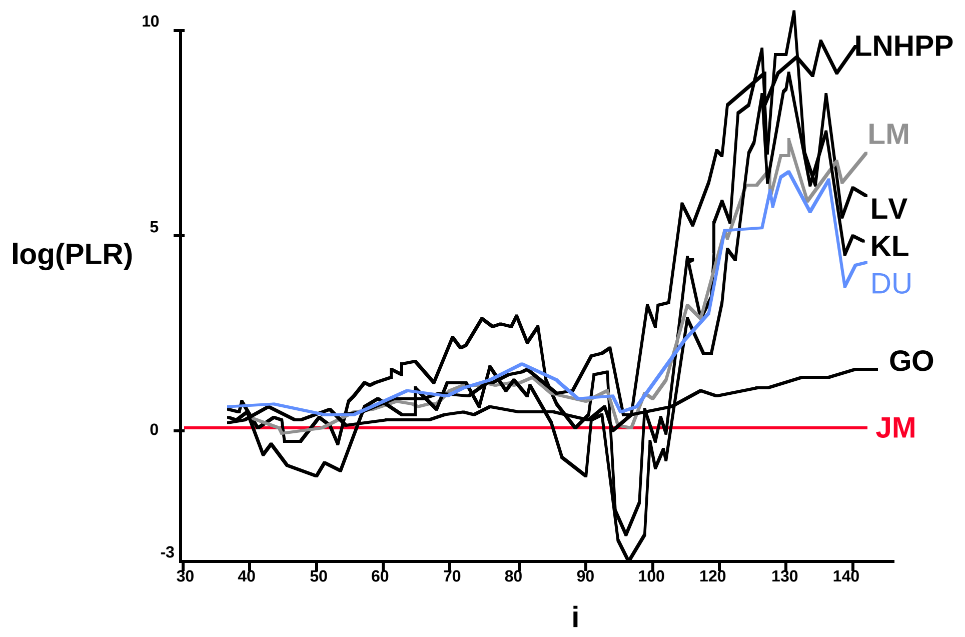


Figure iii: Log(PLR) versus i for the same data

Figure iii shows a PLR comparison of the predictions of the models. Here JM has been chosen as the reference model, and pair-wise comparisons with it carried out for the other models. This confirms the observation we made of the median predictions, that there is little to choose between the earlier predictions - there is no obvious trend in these plots in the early stages. However, from about *i*=95 onwards there is a clear trend upwards for all models except GO, indicating their superiority over JM. At the very end, there is slight evidence of a downward trend in LM, LV, KL and DU, but this is not persistent enough to be certain.

On balance, if one had to select a single model for the next one-step-ahead predictions, LNHPP looks the best candidate. Of course, such decisions as to which model to use for future predictions would be carried out dynamically, using frequent analyses of the kind conducted here as more and more data becomes available.

C.3.5. Discussion

As has been stated earlier, the experience with software reliability growth models is that no single model can be trusted to give universally accurate predictions, and no model can be selected *a priori* in the confidence that it will be accurate on a particular data set. It is for this reason that the methods described above were developed. They enable the analyst to select the model that is performing well (if any is) on the particular data under examination, and thus allow him or her to have reasonable confidence in the results.

The models we have described all base their predictions upon maximum likelihood estimators of their parameters; these estimates are substituted into appropriate mathematical expressions derived from the models in order to predict reliability. This is a somewhat *ad hoc* procedure: it would be better to carry out a proper Bayesian analysis, since there is an exact theory of Bayesian prediction [Aitchison 1975]. Unfortunately, this is presently too difficult to conduct exactly even for the simplest model, although there may be progress in accurate approximations in the coming years as a result of recent research [Smith 1993].

It is similarly impossible to compute exact confidence bounds for the parameters of most of the popular software reliability growth models, even when, as here, simple maximum likelihood estimation is involved. Confidence bounds on the reliability predictions that are computed from these parameter estimates are, of course, even more difficult to compute and are thus generally not available.

This problem is not as serious as it might at first seem. Whilst the techniques such as u-plots and PLR do not allow the accuracy to be quantified in terms of confidence bounds for the predictions, they *do* address the important issues concerning the possible inaccuracy of the underlying *model* being applied. That is, they are based upon a comparison between prediction and actuality, and are thus sensitive to *all* sources of possible inaccuracy, rather than just the sampling accuracy that would be involved in estimation of the model parameters.

C.3.6. Bayesian versus classical frequentist analysis of failure data

As was stated earlier, the reliability growth models, because of their comparative complexity, pose difficulties for Bayesian statistical inference to estimate the model parameters. As a result, estimation of their parameters tends to be conducted via maximum likelihood, and the consequent predictions are then obtained by substitution these point estimates into mathematical expressions obtained from the models. Whilst this is a perfectly reasonable procedure, it does not have the formality of the Bayesian predictive approach. Most importantly, mere substitution of point estimates into these mathematical expressions does not take into account our confidence in the expressions. In contrast, the Bayesian predictive distribution for an unobserved random variable, say the next time to failure T_i, after having the inter-failure times $t_1, . . t_{i-1}$, is the *proper conditional distribution* $p(t_i| t_1, . . t_{i-1}$, model, prior beliefs). In other words, given that the model is correct, the Bayesian procedure gives ‘your’ true distribution for the random variable of interest, and your confidence in the inference about the model parameters is incorporated via the posterior distribution for these parameters.

In general, the Bayesian result will differ from the classical one, although it is sometimes possible to contrive to choose a prior distribution that ‘mimics’ the classical approach. Indeed, a plausible "ignorance" prior, chosen to allow the data to ‘speak for themselves’, can sometimes give results that coincide exactly or nearly exactly with the classical results.

Given these advantages of the Bayesian approach, it should be used whenever possible. For the simpler situation of estimating reliability when there is no reliability growth, a complete Bayesian analysis is feasible, as presented earlier.