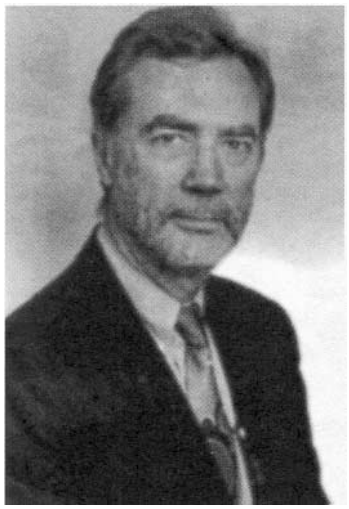
Software Reliability and Dependability: a Roadmap

Bev Littlewood & Lorenzo Strigini

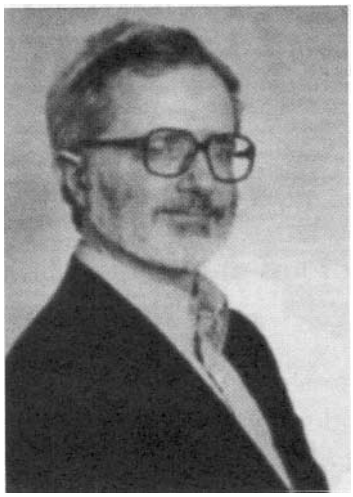
Key Research Pointers

- Shifting the focus from software reliability to user-centred measures of dependability in complete software-based systems.
- Influencing design practice to facilitate dependability assessment.
- Propagating awareness of dependability issues and the use of existing, useful methods.
- Injecting some rigour in the use of process-related evidence for dependability assessment.
- Better understanding issues of diversity and variation as drivers of dependability.

The Authors



Bev Littlewood is founder-Director of the Centre for Software Reliability, and Professor of Software Engineering at City University, London. Prof Littlewood has worked for many years on problems associated with the modelling and evaluation of the dependability of software-based systems; he has published many papers in international journals and conference proceedings and has edited several books. Much of this work has been carried out in collaborative projects, including the successful EC-funded projects SHIP, PDCS, PDCS2, DeVa. He has been employed as a consultant to industrial companies in France, Germany, Italy, the USA and the UK. He is a member of the UK Nuclear Safety Advisory Committee, of IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, and of the BCS Safety-Critical Systems Task Force. He is on the editorial boards of several international scientific journals.



Lorenzo Strigini is Professor of Systems Engineering in the Centre for Software Reliability at City University, London, which he joined in 1995. In 1985-1995 he was a researcher with the Institute for Information Processing of the National Research Council of Italy (IEI-CNR), Pisa, Italy, and spent several periods as a research visitor with the Computer Science Department at the University of California, Los Angeles, and the Bell Communication Research laboratories in Morristown, New Jersey. He holds a "Laurea" cum laude in Electronic Engineering from the University of Pisa, Italy (1980). His research has addressed fault-tolerance in multiprocessor and distributed systems, protocols for high-speed networks, software fault tolerance via design diversity, software testing and software reliability assessment. He has been a principal investigator in several national and collaborative European research projects on these topics, and a consultant to industry on fault-tolerance and on reliability assurance for critical applications. He has published more than 60 papers in international journals and conferences. His main current interest is defining practical, rigorous methods for assessing the dependability of software and other systems subject to design faults, and for supporting development decisions to achieve it. He is a member of the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, of IEEE and ACM and of the Editorial Board of the IEEE Transactions on Software Engineering.

Software Reliability and Dependability: a Roadmap

Bev Littlewood **Lorenzo Strigini**
Centre for Software Reliability, City University
Northampton Square, London EC1V OHB, UK
+44 20 7477 8420 +44 20 7477 8245
b.littlewood@csr.city.ac.uk l.strigini@csr.city.ac.uk

ABSTRACT
Software's increasing role creates both requirements for being able to trust it more than before, and for more people to know how much they can trust their software. A sound engineering approach requires both techniques for producing reliability and sound assessment of the achieved results. Different parts of industry and society face different challenges: the need for education and cultural changes in some areas, the adaptation of known scientific results to practical use in others, and in others still the need to confront inherently hard problems of prediction and decision-making, both to clarify the limits of current understanding and to push them back.

We outline the specific difficulties in applying a sound engineering approach to software reliability engineering, some of the current trends and problems and a set of issues that we therefore see as important in an agenda for research in software dependability.

Keywords
Reliability engineering, dependability modelling and assessment, COTS reliability, diversity

1 INTRODUCTION
We use "dependability" [17] informally to designate those system properties that allows us to rely on a system functioning as required. Dependability encompasses, among other attributes, reliability, safety, security, and availability. These qualities are the shared concern of many sub-disciplines in software engineering (which deal with achieving them), of specialised fields like computer security, and of reliability and safety engineering. We will concentrate on the aspects that are the traditional concern of these last allied disciplines, and will mainly discuss reliability, but many of our remarks will also be of relevance to the other attributes.

In this area, an important factor is the diversity of "the software industry", or, rather, among the many industrial sectors that produce or use software. The demand for software dependability varies widely between industrial sectors, as does the degree of adoption of systematic approaches to it. From many viewpoints, two extremes of the range are found in mass-marketed PC software and in safety-critical software for heavily-regulated industries. A couple of decades ago there was a revolution in dependability of consumer goods such as TV sets, VCRs and automobiles, when companies realised that there was market advantage to be gained by demonstrating higher reliability than their competitors. There has not yet been a similar movement in the corresponding sectors of the software industry.

1.1 Why is our dependence on software increasing?
It is commonplace that software is increasingly important for society. The 'Y2K bug' has just brought this to the attention of the public: not only was a huge expense incurred for assurance (verification and/or fixes) against its possible effects, but this effort affected all kinds of organisations and of systems, including many that the public does not usually associate with computer software. It is useful to list various dimensions of this increased dependence:

- *Software-based systems replace older technologies in safety- or mission-critical applications.* Software has found its way into aircraft engine control, railroad interlocking, nuclear plant protection, etc. New critical applications are developed, like automating aspects of surgery, or steering and 'piloting' of automobiles. Some of these applications imply 'ultra-high' dependability requirements [24]. Others have requirements that are much more limited, but require the development of a computer dependability culture in the vendors (e.g., equipment manufacturers without previous experience of using computers in safety-critical roles) and/or in the customers and users (e.g., doctors and surgeons);
- *Software moves from an auxiliary to a primary role in providing critical services.* E.g., air traffic control systems are being modernised to handle more traffic, and one aspect of this is increasing reliance on software. The software has traditionally been regarded as non safety-critical, because humans using manual backup methods could take over its roles if it failed, but increasing traffic volumes mean that this fall-back capability is being eroded. Here the challenge is to evolve a culture that has been successful so far to cope

Future of Sofware Engineering Limerick Ireland

with higher dependability requirements under intense pressure for deploying new systems;

- *Software becomes the only way of performing some function which is not perceived as 'critical' but whose failures would deeply affect individuals or groups.* So, hospitals, supermarkets and pension offices depend on their databases and software for their everyday business; electronic transactions as the natural way of doing business are extending from the financial world to many forms of 'electronic commerce';
- *Software-provided services become increasingly an accepted part of everyday life without any special scrutiny.* For instance, spreadsheet programs are in widespread use as a decision-making aid, usually with few formal checks on their use, although researchers have found errors to be extremely frequent in producing spreadsheets [27] and the spreadsheet programs themselves suffer from many documented bugs and come with no promise of acceptable reliability by their vendors;
- *Software-based systems are increasingly integrated and interacting, often without effective human control.* Larger, more closely-coupled system are thus built in which software failures can propagate their effects more quickly and with less room for human intervention.

With increased dependence, the total societal costs of computer failures increase. Hence there is a need to get a better grip on the trade-offs involving dependability, in many cases to improve it and generally better to evaluate it.

1.2 Why is there a problem with software reliability?

The major difference between software and other engineering artefacts is that software is pure design. Its unreliability is always the result of design faults, which in turn arise from human intellectual failures. The unreliability of hardware systems, on the other hand, has tended until recently to be dominated by random physical failures of components - the consequences of the 'perversity of nature'. Some categories of hardware systems do fail through design and manufacturing defects more often than is desirable - for example buildings in poor countries - but engineering knowledge is sufficient, at least in principle, to prevent these systematic failures.

Reliability theories have been developed over the years which have successfully allowed hardware systems to be built to high reliability requirements, and the final system reliability to be evaluated with acceptable accuracy. In recent years, however, many of these systems have come to depend on software for their correct functioning, so that the reliability of software has become more and more important.

The increasing ubiquity of software stems, of course, from its general-purpose nature. Unfortunately, however, it is precisely this that brings disadvantages from the point of view of achieving sufficient reliability, and of demonstrating its achievement. Rather informally, these problems stem from the *difficulty* and *novelty* of the problems that are tackled, the *complexity* of the resulting solutions, the need for short development timescales, and finally the difficulty of gaining *assurance* of reliability because of the inherently discrete behaviour of digital systems.

Novelty

Whereas in the past computer-based systems were often used to automate the solution of problems for which satisfactory manual solutions already existed, it is increasingly common to seek computerised solutions for previously unresolved problems - often ones that would have been regarded as impracticable using other technology. This poses particular difficulties for systems with high reliability requirements, since it means that we can learn little from experience of previous systems. Other branches of engineering, by contrast, tend to have a more continuous evolution in successive designs. The change itself to a software-based system, for example from a non-digital electronic control system, may be regarded as a step change in technology. Equivalent step-changes in other branches of engineering are known to be risky, for example the attempt to introduce new materials for turbine blades that led to insolvency and nationalisation for Rolls Royce in 1971.

Difficulty

There is a tendency for system designers to take on tasks that are intrinsically *difficult* when building software-based systems. Software frees the designer from some of the constraints of a purely hardware system, and allows the implementation of sometimes excessive extra functionality.

Thus there are examples of software being used to implement difficult functionality that would be inconceivable in older technologies - e.g., enhanced support to pilots in fly-by-wire and unstable aircraft control, dynamic control of safe separation between trains in 'moving block' railway signalling. Most complex modern manipulations of information - e.g., the control of massive flows of funds around the world's banking systems, or the recent growth of e-commerce - would not be possible without software.

The more difficult and novel the task, of course, the more likely that mistakes will be made, resulting in the introduction of faults which cause system failure when triggered by appropriate input conditions. In the worst cases, the over-weening ambition of designers has resulted in systems being abandoned before they were even complete, with consequent heavy financial loss.

Complexity

Most importantly, these trends toward new and increased functionality in computer-based systems are almost unavoidably accompanied by increased *complexity*. There is no universally accepted measure of complexity, but if we look at simple *size* as a rough-and-ready indicator, its growth is evident - see, for example, the growth in packages such as MS Office from one release to another.

Great complexity brings many dangers. One of the greatest is difficulty of understanding: it is common to have systems that no single person can claim to understand completely, even at a fairly high level of abstraction. This

produces uncertainty about the properties of the program - particularly its reliability and safety.

Control of unwarranted complexity is thus an important aspect of good design: a system should be no more complex than it need be to deliver the needed functionality. Clearly some of the trends discussed above militate against control of complexity. When complexity *is* needed, the challenge is to determine how much the added intellectual difficulty detracts from the dependability of the product.

Assurance

Finally, the inherent discreteness of behaviour of digital systems makes it particularly difficult to gain assurance of their reliability. In contrast to conventional mechanical and electrical systems, it is usually impossible to extrapolate from evidence of failure-free operation in one context in order to claim that a system will perform acceptably in another, similar context. It is, of course, almost always infeasible to test all such contexts (inputs).

Knowing that software is sufficiently reliable is necessary before we can make intelligent decisions about its use. This is clear for safety-critical systems, where we need to be sure that software (and other) failures will not incur unacceptable loss of human life. It is less clear, but we believe also important, in more mundane applications where, for example, it must be decided whether the trade-off between new functionality and possible loss of reliability is cost-effective. There is abundant anecdotal evidence of financial losses from computer undependability: many users need better estimates both of the frequency and of the possible impact of computer failures. This is part of the general need for better assessment of the effectiveness of automation projects.

It is this problem of assurance that has been at the centre of our own research interests; it will thus form a large part of the remainder of the paper.

1.3 Industry demand and concerns

These different factors are common to all software-related industries, but their combinations vary.

The baleful impact of novelty is evident in much of the software used for important everyday tasks, like office automation. This is developed and marketed in ways that are closer to fashion-driven consumer goods than to professional tools. Dependability takes very low priority. New releases are frequent, and tend to include new features to outdo competitors and lure customers into making a new purchase. Reported bugs are preserved in the next release. The user's manual gives an ambiguous description of many functions of the software, and their semantics change between releases, or even between different parts of the same software suite. Many functions are used by small subsets of the user population, making many bugs difficult to find and economically uninteresting to fix. Furthermore, the platforms on which they run often do not enforce separation between the various applications and software supporting them, so that failures propagate, reducing system reliability and complicating fault reporting and diagnosing.

A feature-dominated development culture is part of a competitive situation in which the time-to-market for new features is perceived by producers as the dominant economic driver. Scarcity of expertise in recent technologies for producing software commands high salaries and a premium over experience and reliability culture. Thanks to tools like application-specific languages, libraries of components, spreadsheet and database programming packages, many more people can build complex software-based systems more quickly than was previously possible, often without a formal technical education and without an apprenticeship with more experienced professionals. Compared to more traditional software professionals, these new designers may be as effective at building apparently well-functioning systems, but are unaware of the accumulated experience of risks and pitfalls in software design, and may well lack the required skills of abstraction and analysis.

In this kind of market, both producers and users have little scope for a rational approach to dependability. Vendors do not offer suitable information for comparing products. The reliability of any one application, besides varying with the way it is used (the relative frequencies of the different types of functions it is required to perform and of the inputs to them), depends heavily on the other applications with which it coexists in the same computer. Even for performing very simple tasks we depend on complex software (e.g., to add two columns of numbers we may use a feature-rich spreadsheet program) and hence we obtain lower reliability than we could.

Last but not least, cultures have developed in which excessive computer undependability is accepted as, and thus becomes, inevitable. Users of office software, for instance, often perceive the software's behaviour as only approximately predictable. They are often unable to discriminate between their own misunderstandings of procedures and genuine software failures and often blame themselves rather than designers of poor or poorly documented systems [26].

At the other end of the spectrum, software for safety-critical application is subject to stringent development procedures intended to guarantee its reliability. Costs are much higher, times-to-market longer, innovation slower. Competitive pressures on these factors are resisted by a necessary conservatism in the regulator, the customers and/or the developers. However, little is known about the actual reliability gains from the various assurance techniques employed, about the actual reliability of new (and often even of mature) products, and about the dependability penalties implied by novel, complex applications or new features. When regulators lack confidence about the reliability of a new product, licensing delays may ensue with huge costs. Different industrial sectors adhere to different standards, creating barriers between markets. Some of these differences may be mere historical accidents, but there is little scientific knowledge to support a choice between the alternative practices that they prescribe. It is then natural for each sector to cling to its own, apparently satisfactory practices.

2 WHY PROBABILISTIC RELIABILITY?

People who are new to the problems of software reliability often ask why reliability needs to be described in terms of probabilities. After all, there is a sense in which the execution of a program is completely deterministic. It is either fault-free, in which case it will never fail; or it does contain faults, in which case any circumstances that cause it to fail once will *always* cause it to fail. This contrasts with hardware components which will *inevitably* fail if we wait long enough, and which can fail randomly in circumstances in which they have previously worked perfectly.

Reliability engineers often call failures due to software (and other failures arising from design defects) *systematic*, to distinguish them from *random* hardware failures. This is somewhat misleading: it suggests that in the one case using probabilities is inevitable, but that in the other we might be able to get away with completely deterministic arguments. In fact this is not so, and probability-based reasoning seems inevitable in both cases. The word *systematic* for software failures really refers to the fault mechanism, i.e. the mechanism whereby a fault reveals itself as a failure, and not to the failure *process*. It is correct to say that if a program failed once on a particular input (i.e. particular set of input and state values *and* timings) it would always fail on that input until the offending fault had been successfully removed, and the term 'systematic' describes this, rather limited form of determinism.

However, we are really interested in the failure *process*: what we see when using the system under study - and in particular the software. The software failure process arises from the *random* uncovering of faults during the execution of successive inputs. We cannot predict with certainty what all future input cases will be and we do not know the program's faults. So, we would not know which inputs, of the ones we had not yet executed, would produce a failure if executed (if we did know this, we could use the information to fix the fault).

So, there is inevitable uncertainty in the software failure process, for several reasons. This uncertainty can only be captured by probabilistic representations of the failure process: *the use of probability to express our confidence in the reliability of a program is therefore inevitable*. The language and mathematics of reliability theory are as appropriate (or inappropriate) for dealing with software reliability as they are for hardware and human reliabilities. In particular, it is appropriate, during the construction of a system, to assign a probabilistic reliability target even though, in the most general case, the system is subject to random hardware failures, human failures, and failures as a result of software or hardware design faults.

3 WHAT LEVELS OF RELIABILITY ARE CURRENTLY ACHIEVABLE?

The difficulty of achieving and demonstrating reliability depends on the level of reliability that is required. This varies quite markedly from one application to another, and from one industry to another, but should rationally be determined by considering the cost of the consequences of failures. Some of the most stringent requirements seem to apply to applications involving active control. For instance, software-based flight control systems ('fly-by-wire') in civil airliners fall under the requirement that catastrophic failures be "not anticipated to occur over the entire operational life of all airplanes of one type", usually translated as 10^{-9} probability of failure per hour [5]. By contrast, safety systems (systems that are only called upon when some controlled system gets into a potentially dangerous state) such as nuclear reactor protection systems, often have relatively modest requirements: for example, some nuclear protection systems have a requirement of 10^{-4} probability of failure upon demand (*pfd*).

The most stringent of these requirements look extremely difficult to satisfy, but there is some evidence from earlier systems that very high software reliability has been achieved during extensive operational use. Reliability data for critical systems are rarely published, but, for instance, measurement-based estimates on some control and monitoring systems give a failure rate of $4 \cdot 10^{-8}$ per hour for potentially safety-related functions [18]. An analysis [30] of FAA records (while pointing at the extreme difficulty of extracting trustworthy data) tentatively estimated failure occurrence rates in avionics software to vary in the range 10^{-7} to 10^{-8} for those systems in which failures prompted the issue of FAA 'airworthiness directives'. The AT&T telephone system historically exhibited very high quality-of-service measures, achieved by focusing not only on component reliability but also extensive redundancy, error detection and recovery capabilities. For instance, the 4ESS switches achieved observed downtime (from all causes) of less than 2 hours per 40 years, or about $5.7 \cdot 10^{-6}$ unavailability [4]; a recent analysis [15] indicates that software failure accounts for only 2% of telephone service outage time experienced by customers.

It is interesting, but perhaps not surprising, that hard evidence about achieved levels of software reliability come from those industries where the required levels are extremely high: typically these industries have reliability cultures that long preceded the introduction of computer systems. Figures from the newer IT industries are much harder to come by. However, there is much anecdotal evidence of low reliability from the users of PC software, and this viewpoint has not resulted in any authoritative rebuttal from the industry itself.

It should be emphasised that the evidence, above, of having achieved extremely high reliability was only available *after the event*, when the systems had been in operational use for extremely long times. In fact for most of these systems, particularly the safety critical ones, the assurance that the reliability target has been met is needed *before* the systems are deployed. This remains one of the most difficult problems in software reliability.

4 HOW CAN WE MEASURE AND ASSURE RELIABILITY?

We now consider briefly the different types of evidence that can support pre-operational claims for reliability. In

practice, particularly when high levels of reliability need to be assured, it will be necessary to use several sources of evidence to support reliability claims. Combining such disparate evidence to aid decision making is itself a difficult task and a topic of current research.

4.1 Testing of software under operational conditions

An obvious way to estimate the reliability of a program is to simulate its operational use, noting the times at which failures occur. There has been considerable research on the statistical techniques needed to analyse such data, particularly when faults are removed as they are detected. This *reliability growth modelling* [2, 25] is probably one of the most successful techniques available: it is now generally possible, given the availability of appropriate data, to obtain accurate estimates of reliability *and to know that they are accurate.*

There are, however, limitations to this approach. In the first place, it is often difficult to create a testing regime that is statistically representative of operational use. For reliability assessment, doubts will remain on whether inaccuracies in the testing regime may invalidate the reliability predictions obtained. In some areas - e.g., general office products, management information systems - the products often change the way in which their users operate so that the operational environment is not stable.

Secondly, the reliability growth models tend to assume that fault removal is successful: they can be thought of as sophisticated techniques for trend fitting. They will not capture any short-term reversals of fortune, such as a failure to remove a fault or, worse, the introduction of a new fault. This has serious implications in critical applications, where the possibility that the last fix might have introduced a new fault may be unacceptable. This is the case in the UK nuclear industry, for example, where the conservative assumption is made that any change to a program creates a *new* program, which must demonstrate its reliability from scratch.

Finally, the levels of reliability that can be assured from these kinds of data are quite limited. To demonstrate a mean time between failures of x time units using the reliability growth models can require a test of duration several hundred times x time units [24]. Similarly, if we seek a conservative assessment by only considering testing after the last change to the software, achieving, e.g., 99% confidence in 10^{-3} *pfd* would require 4600 statistically representative demands to be executed without failures; 99% confidence in 10^{-4} would need 46000 demands without failure, and so on. Increasing the reliability level to be demonstrated increases the length of the test series required until it becomes infeasible. Yet, these methods are adequate for the levels of reliability that are required of many practical systems (*cf* the safety systems quoted above).

4.2 Evidence of process quality

Since it is obvious that the quality of a process affects the quality of its product, it is accepted practice that the higher the dependability requirements for a system, the more stringent quality requirements are imposed on its development and validation process. For instance, standards for software for safety-critical systems link sets of recommended or prescribed practices to the level of required reliability. The fact of having applied the recommended practices is then often used as a basis for claiming that the corresponding reliability level has been achieved. Unfortunately, there is no evidence that the former implies the latter. In a parallel development, in recent years there has been increasing emphasis on the contribution of strict control on the software development process to product quality. But again, although common sense tells us that it is unlikely for poor development procedures to produce highly reliable software, there is little or no evidence indicating how much benefit can be expected from the use of good process. Indeed, it is clear that good process can sometimes result in very unreliable products. Even if we had extensive experience of the relationship between process and product qualities on previous products, it seems likely that this will contain large statistical variation, and thus preclude strong conclusions being drawn about a particular new product.

There are similar problems in relating counts (or estimates) of software faults to reliability. Even if we could trust the statistical techniques that estimate the numbers of faults left in a program, which is doubtful [9], it is not possible to use this information to obtain accurate reliability predictions. One reason for this is that the 'sizes' of software faults seem to be extremely varied [1]: to know the reliability of a program it is necessary to know both the number of faults remaining and the contribution that each makes to unreliability.

4.3 Evidence from static analysis of the software product

Static analysis techniques clearly have an important role in helping to achieve reliability. It also seems intuitively obvious that they could increase *confidence* in the reliability of a program. For example, a formal proof that a particular class of fault is not present in a program should make us more confident that it will perform correctly: but *how much* more confident should we be? More precisely, what contribution does such evidence contribute to a claim that a program has met its reliability target?

At present, answers to questions like this are rather informal. For example, probably the largest Malpas [31] analysis ever conducted was for the safety system software of the Sizewell nuclear reactor. This showed up some problems, but it was claimed that none of these had safety implications. On the other hand, certain parts of the system defeated the analysis tool because of their complexity. Thus while some considerable comfort could be taken from the analysis, the picture was not completely clear. At the end of the day, the contribution of this evidence to the safety case rested on the informed judgement of expert individuals.

4.4 Evidence from software components and structure

For many kinds of non-software systems, 'structural' models of reliability allow the reliability of a system to be

derived from the reliabilities of its components, which are often easier to estimate or known before the system is even built. Achieving a similar ability for software systems is a yet unfulfilled aspiration. 'Structural' models of reliability have been indeed developed for software [3, 16, 21]. They could, in principle, be used when a system is built out of COTS items: the component reliabilities can, in principle, be estimated from their previous operational use in other systems. But practical issues still stand in the way (and will appear again in the next section about 'future challenges'):

- obtaining such data, which are seldom documented with sufficient accuracy and detail to allow confident predictions;
- knowing that the interactions among components are actually limited to those that a simple model can capture; e.g., spurious transfers of control or modules overwriting other modules' variables may be difficult to represent in manageable models;
- estimating the benefits of redundancy, a traditional application of this kind of modelling, is complicated by the need to estimate failure correlation between redundant components. For software systems, it is impossible to assume failure independence even among diverse components [22], and there is no basis for the experience-based, conservative rules of thumb used in other areas of engineering;
- knowing whether the previously measured reliabilities will apply in the novel context, since the reliability of each components depends on its usage profile, which will vary between systems.

5 TRENDS AND RESEARCH CHALLENGES FOR THE FUTURE

Among the challenges that we list here, only some are actually 'hard' technical research topics. The difficulties in applying reliability and dependability techniques in current software engineering are quite often cultural rather than technical, a matter of a vast gap of incomprehension between most software experts and dependability experts. For an actual improvement in engineering practice, it is necessary to bridge this gap. This may require more than just goodwill, but research into its economic, cultural and psychological causes and how to deal with them.

There is a general trend towards greater utilisation of off-the-shelf software, which offers some promise for both better reliability and better ability to assess it. Wider populations of users make the effort of high-quality development and assurance activities more affordable. This does not guarantee that this effort will be made, though: with mass-distributed consumer software, for instance, these economies of scale have been used instead for reducing prices or increasing profits. For dependability-minded customers, like the safety-critical industries, quality of COTS products is now a major concern. Re-use of COTS items may also pose difficulties and reliability risks if, as is common, the components were not designed within a re-use strategy in the first place. This issue is open to empirical study. A large user base should also help with problem reporting and product improvement but, again, this potential would only be realised given sufficient economic incentives.

Widely used off-the-shelf components should also offer the possibility of using data from their previous operational use to extrapolate future reliability and to support the structure-based reliability models described in section 4.4. To allow this extrapolation, characterising differences between usage environments and their effects on reliability becomes an important research problem. Immediate goals could be simply rules for conservative extrapolations from past experience, or about when extrapolation is legitimate, expressed in terms of the characteristics of components, architectures and system use.

The practical difficulties listed should apply less to those software producers that cater to safety-critical applications. Here, there is also a trend towards standardisation and consolidation of product lines, so that developing new applications is increasingly a matter of customisation rather than ad-hoc design. With pressure from the customers, this trend may realise the promises of the 'COTS movement' sooner than in the general market, using the wide diffusion of the same components both to improve the software faster and to measure achieved reliability. A need here is to develop practices for documenting past reliability records that can become accepted standards.

Interestingly, many supporters of the 'open source' approach claim that it produces improved reliability. It is difficult to verify these claims and, assuming they are correct, to clearly account for the causes of higher reliability and determine to how wide a range of products they could be extended. Tapping the expertise of users for diagnosing and even repairing faults is attractive. Customers with high reliability requirements may mistrust the apparently lax centralised control in the open-source process, but even for them disclosing source code offers more informed bug reporting and distributed verification. In a related area, many security experts believe that using secret algorithms is often a self-defeating move for designers, as it deprives them of the advantage of scrutiny by the research community. Clarifying the advantages and disadvantages of the various aspects of the open-source approach on an empirical basis, and, more modestly, exploiting it as a source of data for other reliability research, are two necessary items on the agenda of research in software dependability.

5.1 Propagating awareness of dependability issues and the use of existing, useful methods

It is common for computer scientists to complain about the poor quality of current software, and for vendors to reply that their choices are dictated by their customers. Without judging where the truth lies between these somewhat self-serving positions, it seems clear to us that society would benefit from greater awareness of software dependability problems. There is room for great improvements among users - both end users and system integrators.

Public perception of software dependability

On new Year's Day, 2000, media reports proclaimed that very few computer systems had failed, and thus the huge 'Y2K' expenditure had been wasted. These reports show

ignorance of a few facts. Computer failures need not be immediately obvious, like 'crashes'. They may be hard to detect; knowing the approximate form of a software fault (a 'Y2K' fault) does not mean knowing when it will cause a failure. Since computers are state machines, they may store an erroneous state now which will cause a failure after a long time of proper service. Last but far from least, knowledge about dependability is always uncertain, and investing in reducing this uncertainty is often worthwhile. Increased awareness of these issues would certainly allow users better to address system procurement, to prepare and defend themselves against the effects of failures, and to report problems and requirements to vendors.

Design culture

With the blurring of the separation between professional software developers and users, these misperceptions increasingly affect system development. But even professional developers often lack education in dependability, both from academic learning and from their workplace environment. The RISKS archives (http://www.CSL.sri.com/risksinfo.html) are a useful source for new and old developers, users and educators. They document both useful lists of common problems, for those who wish to learn from historical memory, and the lack of this shared memory for many users and developers. Many reported problems stem from repeated, well-known design oversights (e.g., 'buffer overflow' security vulnerabilities). The same cultural problems show up again and again: lack of risk analysis and of provisions of fall-backs and redundancy, focus on a technical subsystem without system-level consideration of risks.

Management culture

Assessing dependability and taking high-level engineering decisions to achieve it run into different problems. Here we deal with uncertainty, requiring understanding of probability and statistics applied to rather subtle questions. Managers who come from software design do not usually have an appropriate background. The errors in applying theoretical results to decision-making are often very basic: ignoring the limits of the methods (e.g., accepting clearly unbelievable prediction of ultra-high reliability [20], or trusting failure probability estimates to multiple significant digits), misusing one-number measures (e.g., using an MTTF comparison to choose a system for which the main requirement is availability over short missions: a serious error), embracing methods from the scientific literature which have been proven inadequate (e.g., requiring a vendor to estimate reliability by a specific method that errs in favour of the vendor). The knowledge that decision-makers need concerns the basic concepts of dependability and uncertainty and awareness of the misunderstandings that arise between the software and the reliability specialists. Perhaps the most serious challenge for the reliability engineer is in delimiting the role for the probabilistic treatments of dependability: on the one hand, clarifying the limits of the possible knowledge of the future; on the other hand, pointing out that if we really want to examine what we know, some formalism is an indispensable support for rigorous thought. In some industries, labels like "10^{-9} probability of failure" are now applied without much consideration of what evidence would really be required for claiming them. In practice, this probabilistic labelling is a conventional exercise, even where there is the most serious attention to safety. The challenge is to make practitioners accept that a well-founded claim of "better than 10^{-4}" would be more useful to them, and to make the public accept that this is not a change for the worse.

5.2 Focus on User-Centred, System-Level Dependability Qualities

All too often, reliability is described in terms of compliance of a specific program with its written specifications. This may have paradoxical consequences: if a program was written with imprecise, largely unstated requirements, does this imply that we cannot state reliability requirements for it? The sensible way of approaching reliability is to define failure in terms of a system's effect on its user. For instance, in using a computer to write this article, we have a very clear perception of what would constitute a failure, e.g., the computer reacting to a command with an unexpected change to the text, or its crashing or corrupting a stored file. Measuring the reliability of individual components with respect to component-specific requirements is, in other areas of engineering, a convenient step towards assessing the satisfaction of the user's dependability requirements (*cf* 4.4). It may also be useful for carefully structured software-based systems, ones in which, for instance, altering the options for my E-mail-reading software cannot destroy recent changes to my article. But component-based assessment is not the goal of reliability engineering. For the user, failures are classified by their consequences rather than their causes: it does not matter to me whether I lose my article because the word processor contains a bug, or because the platform allows an unrelated application to interfere with the word processor, or because the manual to the word processor does not explain the side-effects of a certain command. Actually, most users cannot tell whether a certain undesired behaviour of a word processor is due to a bug or to their misunderstanding of the function of the software. The system I am using to produce the printed article includes the computer with its software as well as myself, and it is the reliability of this system that should be of concern to designers.

User-oriented requirements have many dimensions. Thus, traditionally, telephone companies established multiple target bounds for the frequency of dropped calls, frequency and total duration of outages, and so on. Users of an office package have distinct requirements regarding the risks of corruption to stored data, of unintended changes to an open file, or interruptions of service. All these needs are served by attention to various aspects of design: reliability of application modules, robustness of the platform, support for proper installation and detection of feature interactions, effective detection of run-time problems, informative error messages and design to facilitate recovery by the user.

With an accent on integration rather than ex-novo design, and a climate of feature-dominated frequent improvements, most system integrators and users find themselves using software whose reliability is difficult to assess and may turn

out to be very poor in their specific environments. This increases the importance of resilience or fault tolerance: the ability of systems to limit the damage caused by any failure of their components. Propagating a culture of robust design, and exploring its application in modern processing environments, seems an essential part of improving dependability in the short term. Measuring robustness is essential for trusting systems built out of re-used components. Examples of attempts in this direction are [6, 13], but there are still challenges in studying how to obtain robust or conservative estimates given the unknown usage pattern to which the software may be subjected.

In all these areas, dependability in software in general could benefit from lessons learned in the area of safety, e.g., the need for systematic analysis of risks ('hazards' for the safety engineer) early on during specification and of prioritising dependability demands, the realisation that maintenance and transition phases are an essential and critical part of a system's life, the importance of human factors in both operation and maintenance [19, 29], the need to understand the genesis of mistakes, the necessity of fault tolerance (error detection and recovery) and of diversity.

5.3 Design for dependability assessment

The difficulties in assessing software dependability are due in part to the complexity of the functions that we require from software, but also for a large part to design cultures that ignore the need for validation. Engineers have traditionally accepted that the need to validate a design (to demonstrate beforehand that the implemented system will be serviceable and safe) must constrain design freedom. Structures have been limited to forms that could be demonstrated to be acceptably safe, either by extensive empirical knowledge or by the methods of calculation known at the time. The less a new design could be pre-validated on models and prototypes, the more conservative the design had to be. This restraint has been lost in a large part of the software industries. We list here design practices that have a potential for facilitating validation and thus reliability engineering.

Failure prevention

A generally useful approach is that of eliminating whole classes of failures. One method is proving that certain events cannot happen (provided that the software implementation preserves the properties of the formal description on which the proof is based). Another set of methods uses the platform on which software runs to guarantee *separation* of subsystems. Memory protection prevents interference and failure propagation between different application processes. Guaranteed separation between applications has been a major requirement for the integration of multiple software services in few powerful computers in modern airliners. We recommend [14] for a thorough discussion of separation and composability.

It should be noted that these methods can support one another. E.g., hardware-level separation between applications prevents some departures from the behaviour assumed in formal proofs of 'correctness' based on high-level descriptions. Exploiting this synergy for dependability assessment is a possibility that has not been explored, although a suitable approach is described in [28].

These methods favour dependability engineering in multiple ways. First of all, they directly increase reliability by reducing the frequency or severity of failures. Run-time protections may also detect faults before they cause serious failures. After failures, they make fault diagnosis easier, and thus accelerate reliability improvements. For dependability assessment, these failure prevention methods reduce the uncertainties with which the assessor has to cope. The probability of any failure to which they apply can be trusted to be lower than the probability of, e.g., an error in a proof or a failure of a hardware protection mechanism - often negligible in comparison to the probabilities of other software failure modes. So, for instance, sufficient separation between running applications means that when we port an application to a new platform, we can trust its failure rate to equal that experienced in similar use on a previous platform plus that of the new platform, rather than being also affected by the specific combination of other applications present on the new platform. This is a step towards applying structure-based reliability models (*cf* Section 4.4). Some difficulties typical of software would remain (failure dependence between subsystems, wide variation of reliability with the usage environment), but the range of applicability of structure-based models would certainly increase.

System monitoring

Testing for reliability assessment can also be aided by software designers. They can simplify the space of demands on the software which testers need to sample, and simplify the set of variables that the test designer must understand in order to build a realistic sample of the usage profile of the software. For instance, periodic resets limit the traces of operation of the software to finite lengths; subsystem separation reduces the number of variables affecting the behaviour of each subsystem; elements of defensive and fault-tolerant programming - assertions for reasonableness check, interface checks, auditing of data structures - improve the ability to detect errors and failures, so that failure counts from testing becomes more trustworthy (cf [11]).

Error detection techniques have an important role throughout the lifetime of systems. No matter how thoroughly a system has been assessed before use, information from its actual failure behaviour in use is precious. Reported errors and failures can lead to faults being corrected. For instance, the civil aviation industry has procedures for incident reporting and promulgation of corrections to equipment and procedures that contribute to its general safety. Besides improving dependability, monitoring is useful for improving its assessment. For instance, when a safety-critical system starts operation, the assurance of its being sufficiently safe is affected by various uncertainties. Even if it has been tested in realistic conditions, a prediction on the probability of future accidents is only possible with rather wide bounds, due to both the possibility that actual use will differ from predicted use, and to the fact that the period of test was limited. As operation continues, both factors of uncertainty are reduced

(in a way that is easily captured by mathematical formulations for the latter, and requires more ad hoc, informal reasoning for the former).

Monitoring requires a *technical* component - effective means for automatically detecting and logging problems - and an *organisational* component - procedures and incentives for the data thus logged to be collected AND analysed. The technical means have been around for a long time. The organisational part is more difficult. Experience teaches that a vendor's dedication may not be enough, as users may be selective in reporting failures. However, given a will, a vendor of even, say, personal computer operating systems could reach the point of being able to advertise the reliability of the operating system using truthful measurement from the field. The technical means are there.

All these approaches come together when we consider the 'COTS problem'. When integrating a COTS subsystem in a new system with explicit dependability requirements, it would seem natural for a designer to require assurance in some appropriate form: possibly, a documented proof of correctness from specified viewpoints, and certainly an indication of the forms of monitoring applied in previous uses and the reliability data thus collected. Thus, for instance, the price of COTS components could increase with documented experience as the risk of using them decreases, allowing more efficient cost-effectiveness decisions for dependability-minded designers.

5.4 Diversity and variation as drivers of dependability

'Functional diversity' or, less frequently, 'design diversity' are common approaches to ensuring safety or reliability of critical systems. The study of their effectiveness is still open: while they have been shown to deliver reliability improvements, the evidence about their cost-effectiveness and their limits, compared to other techniques, is still as primitive as for most software engineering techniques. Further study is certainly warranted, in particular given the increasing pressure to use COTS items. Integrators using these components have limited control on their quality. Given the alternative between using diverse COTS systems in a redundant configuration and procuring a higher-quality, bespoke system, the former alternative may often be the only way of achieving a modest level of assurance at reasonable cost.

A more important consideration has emerged from the study of design diversity, i.e., that some form of diversity is a factor in most methods for ensuring dependability. For instance, we inspect software to find those faults that were not avoided in writing it. We combine two forms of verification in the hope that one of them may be effective on those faults that the other failed to find. The usefulness of a technique is then a function of *both* its effectiveness if used alone (e.g., the ability of a testing technique to detect faults and thus lead to improved reliability), *and* of how well it complements the rest of the process in which it is applied. Between two techniques of equal cost, the one that is proven more effective in isolation may turn out to be the less effective in combination with the rest of the process. This trade-off is outlined in [23], and implies a need for extending empirical research from measuring the 'absolute' reliability potential of software engineering techniques in isolation to either characterising whole processes or characterising the similarities and differences between the effects of the individual techniques.

5.5 Judgement and decision-making

Engineering approach

Software reliability research is often seen as the production of techniques for increasing reliability. This view often prompts the response that industry cannot afford any extra effort on reliability. This is a limiting view. Engineering should deal with *control*. In achieving dependability, this means matching the means to the dependability targets. In assessing it, it means predicting some effects of using the assessed system. It is true that predicting the reliability that a software project will achieve is affected by large uncertainties. These must be taken into account for rational decisions. In particular, overkill is a sensible strategy if the costs of overkill in process quality are outweighed by the potential extra costs of undependable products.

However, competition militates against overkill in seeking dependability. For instance, in the telecoms industry, increasing deregulation frees service providers to seek different trade-offs between cost and quality of service, of which software dependability is a component. This encourages more precise tuning of expenditure on reliability to the perceived needs: witness, for instance, the continuing success of the IEEE conference on software reliability engineering, ISSRE. The topics debated include the feasibility of new techniques, but also attempts to measure and thus compare the actual advantages and costs of alternative techniques. This activity is important and especially discriminating any common lessons from the multitude of individual studies is a challenging topic with large potential rewards.

There are legitimate concerns with possible over-confidence in a reliability engineering approach. It is tempting, in all areas of engineering, to ask for deterministic rules linking the reliability techniques to the achieved results. This is never realistic, and even less so in software. The effectiveness of the various techniques is both inherently variable and difficult to study, so the cost difference between a mildly optimistic attitude to achieving a dependability target and a mildly conservative attitude is and may remain large. The challenge for research is to reduce this gap, but also to clarify its existence and its consequences for decision-making.

Choice of process for dependability

A first problem arises in choosing software engineering methods. Most proposed methods for improving software reliability have intuitive appeal, but measuring whether their advantages are real *and* justify their cost is seldom attempted [8, 12]. This situation is possibly changing for the better, judging by the software engineering literature. Many companies now have extensive measurement systems, oriented at project management and process improvement. Data collection may well be used both for

better reliability assessment and for better assessment of the methods used, although their emphasis is often on productivity without reference to dependability.

There are limits to such experimental validation. Given the many factors in a software engineering process, it is difficult to trust data from heterogeneous sources. Indeed, successes in statistically understanding the software process seem mostly to occur in large companies with highly repetitive processes. By implication, small companies with their own special practices or application domains, or any fast-changing development process, would not be susceptible to evaluation of their effectiveness. There are definite research problems here. On the one hand, exploiting the available statistics to extract all the useful conclusions they warrant, but avoiding overgeneralisation, poses demanding problems of data analysis. On the other hand, there is the need to base decisions on general laws that determine the effectiveness of the various methods, rather than raw statistical data. Decision-makers often posit such laws, based on their experience, but these are not subjected to scientific scrutiny and indeed different experts have contrasting beliefs.

Formalism and judgement in assessment

The problem just discussed, of judging the reliability effects of choices in the software process, presents itself in dependability assessment as well: process-based evidence is heavily used in dependability assessment. For modest dependability requirements, the statistical techniques now available (*cf* 4.1) would support assessment without recourse to process evidence, but industry is reluctant to apply the required effort in testing and data collection. For some extreme requirements, as discussed before, using all available evidence becomes essential; adherence to a prescribed process is usually a major part of this evidence, but is not a sufficient basis for believing that the software indeed satisfies the dependability requirements. Injecting some rigour in the use of process-related evidence for dependability assessment is an important research challenge.

The difficulties in rigorously assessing reliability have two facets. One is the scarcity of knowledge, e.g. about the actual effect of process on product dependability. The other is the difficulty of understanding how the disparate evidence that we need to use - e.g., logical evidence from formal analyses, evidence about aspects of the design such as fault tolerance, direct measurements from statistical testing, and so on - should be combined into an overall evaluation, preferably quantitative, of reliability. In principle, both can be handled by probabilistic reasoning. Like all formal languages, the language of probability requires some training, but rewards its user with the ability to better dominate complex problems. Proper use of theory would avoid many errors, not least errors of overconfidence in the absence of sufficient data.

On the other hand, matching formal descriptions to the real world requires judgement and prudence that cannot be formalised. Moreover, the usual input to successful applications of probabilities, namely statistical data, is often lacking when we deal with software. This requires, for instance, restraint in using detailed 'structural' models (*cf* section 4.4). Instead, the lure of deducing formally correct results tempts users to trust complex models with little evidence that they apply to reality. We still see even extreme fallacies like system reliability models in which software failure probabilities, for lack of convenient estimates, are set to 0 or to arbitrary 'estimated' values. Informal expert opinions are often substituted both for trusted physical laws and for raw probabilistic inputs to models. Claims in numerical, probabilistic terms can thus be produced casually or used beyond their legitimate scope. While it is unavoidable for decision-makers to use expert judgement, they should be aware of its limits and of the need to challenge and cross-check all the critical bases of a decision.

For combining disparate evidence, Bayesian Belief Nets (BBNs) are a promising candidate [7], but the need for caution remains. The Bayesian formalism enforces a rigorous probabilistic consistency. However, it guarantees conclusions that are only as trustworthy as the evidence items, and the assumed relationship between them, that go into creating a BBN. So, in spite of the explosion of interest in BBNs in recent years, there are still serious limitations to their use, mainly due to problems with their validation. There is considerable ongoing research on these issues, but assessments produced by BBNs must be seen with a critical eye, especially when dealing with critical systems.

5.6 Very Large-Scale Systems

Very large 'systems of systems' are now built by the (often unplanned) aggregation of computer-based systems. This creates concerns that are outside the traditional domains of software and reliability engineering.

In extending the practice of dependability engineering to such systems, one need is for modelling increasingly complex systems [10] without losing the ability to understand and to solve the models. Another is to avoid the risk that factors that were negligible in previous reliability problems become dominant, making the detailed models useless. A warning is sounded by the still frequent practice in reliability and safety estimates of disregarding the effects of software failures when assessing systems that are increasingly software-based. It may be that radically different models from those hitherto used must be integrated with traditional software reliability practice. Software has made it difficult to apply component-based reliability models familiar to hardware engineers and required black-box, sampling-based approaches to measurement and prediction. Likewise, for large-scale integrated systems we may need to borrow modelling practices that have been successful elsewhere, for instance in modelling the evolution of diseases or the propagation of disturbances in electrical distribution networks.

Other factors in these large systems also require a more interdisciplinary approach to their dependability. Deregulation and privatisation are a common trend in many industries that form the national infrastructures. This leads the industries concerned to scrutinise more stringently their

cost-dependability trade-offs. Here, concerns about large-scale societal vulnerabilities may justify demand for effective achievement and assessment of higher levels of reliability than would be required by the individual players. Some of the actors operate in novel or rapidly changing markets, like Internet applications. Dependability costs are seen up front, while losses from undependability lie in the future. In any case, large systems grow without a general design, and their emerging properties determine the dependability observed by their various users. Design, for these systems, is limited to industry's attempts to standardise or otherwise to influence customer behaviour, and to government regulation. The issues to be studied become the interplay of economic incentives, legal environment and technology development and their joint effect on the complex systems.

5.7 Integration with Human Reliability

A clear requirement in most software design is better attention by software designers to factors like the user's cognitive workload and mental models of the software's operation. But there are other areas in which engineering for reliability has to be integrated with 'human factors' and cognitive research. One such area concerns the diversity and complementarity between people and machines: both between users and computer systems, and between developers and mechanised aids. The early history of automation has brought new understanding of the strengths and weaknesses of humans as operators of machinery compared to those of automated machinery, i.e., of humans as designers of automation. This understanding has yet to be internalised by software engineering practice and research. Another useful direction is exploiting the progress of cognitive psychology applied to the mechanisms of human error for understanding the effects of software processes. While much ergonomic research has examined operator errors in dealing with automation, applications to design activities are limited. We should not expect an ability to predict the effectiveness of a software engineering method from first principles, but useful indications for evaluating and improving the reliability implications of software engineering methods.

6 CONCLUSIONS

An engineering approach to design must include dependability aspects. In software, progress in this direction has been slow, but is necessary for more efficient decisions by both individual actors and society. Increasing dependence on software increases the costs of undependability or of not matching dependability to needs. Some current trends, like that towards using more COTS components, create both opportunities and technical challenges for this progress. There are non-technical difficulties to overcome, in terms of education of users and developers and better communication between technical communities. The research challenges include both learning more about the effects of the practices for achieving dependability and learning better to organise knowledge to support judgement and decision-making.

ACKNOWLEDGEMENTS

The authors' work was supported in part by EPSRC grants GR/L07673 and GR/L57296.

REFERENCES

[1] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development 28, 1* (January 1984), 2-14.

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

[3] R. C. Cheung. A User-Oriented Software Reliability Model. *IEEE Transactions on Software Engineering SE-6, 2* (March 1980), 118-125.

[4] G. F. Clement, P. K. Giloth. Evolution of Fault Tolerant Switching Systems in AT&T. In A. Avizienis, H. Kopetz and J.-C. Laprie (Eds.) *The Evolution of Fault-Tolerant Computing*, Springer-Verlag, 1987, 37-54.

[5] FAA. Federal Aviation Administration, Advisory Circular AC 25.1309-1A, 1985.

[6] J.-C. Fabre et al., Assessment of COTS microkernels by fault injection, in *Proc.* DCCA-7 (San Jose, California, USA, January 1999), 25-44.

[7] N. Fenton, M. Neil. *Software Metrics: a roadmap*", in this volume.

[8] N. Fenton, S. L. Pfleeger, R. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software 11, 4* (July 1994), 86-95.

[9] N. E. Fenton, M. Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering 25, 5* (September/October 1999), 675-689.

[10] N. Fota et al., Safety analysis and evaluation of an air traffic control computing system, in *Proc.* SAFECOMP '96 (Vienna, Austria, October 1996), 219-229.

[11] M. J. Harrold. *Testing: a roadmap*", in this volume.

[12] L. Hatton, Programming Languages and safety-Related Systems, in *Proc.* Safety-Critical Systems Symposium (Brighton, U.K., 1995), 49-64.

[13] P. Koopman, J. DeVale, Comparing the robustness of POSIX Operating Systems, in *Proc.* FTCS-29 (Madison, USA, June 1999), 30-37.

[14] H. Kopetz. *Software Engineering for Real-time: a roadmap*", in this volume.

[15] D. R. Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer 30, 4* (April 1997), 31-36.

[16] J.-C. Laprie, K. Kanoun. X-ware reliability and availability modeling. *IEEE Transactions on Software Engineering 18, 2* (February 1992), 130-47.

[17] J. C. Laprie (Ed.) Dependability: Basic Concepts and Associated Terminology. Springer-Verlag, 1991.

[18] A. Laryd, Operating experience of software in programmable equipment used in ABB Atom nuclear I&C application, in *Proc.* Advanced Control and Instrumentation

Systems in Nuclear Power Plants (Espoo, Finland, June 1994), 31-42.

[19] N. G. Leveson. Safeware: system safety and computers. Addison Wesley, 1995.

[20] N. G. Leveson, C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer 25, 7* (July 1992), 18-41.

[21] B. Littlewood. Software Reliability Model for Modular Program Structure. *IEEE Transactions on Reliability 28, 3* (August 1985), 241-246.

[22] B. Littlewood, D. R. Miller. Conceptual Modelling of Coincident Failures in Multi-Version Software. *IEEE Transactions on Software Engineering SE-15, 12* (December 1989), 1596-1614.

[23] B. Littlewood et al. Modelling the effects of combining diverse software fault removal techniques. *IEEE Transactions on Software Engineering* (to appear).

[24] B. Littlewood, L. Strigini. Validation of Ultra-High Dependability for Software-based Systems. *Communications of the ACM 36, 11* (November 1993), 69-80.

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

[26] D. A. Norman. The Design of Everyday Things. Doubleday, 1990.

[27] R. R. Panko, R. P. J. Halverson, Spreadsheets on trial: a survey of research on spreadsheet risks, in *Proc.* HICSS-29 (Wailea, Hawaii, USA, January 1996), 326-335.

[28] D. Powell, Failure Mode Assumptions and Assumption Coverage, in *Proc.* FTCS-22 (Boston, Massachusetts, USA, 1992), 386-395.

[29] J. Reason. Human Error. Cambridge University Press, 1990.

[30] M. Shooman, Avionics Software Problem Occurrence Rates, in *Proc.* ISSRE'96 (White Plains, New York, U.S.A., October/November 1996), 55-64.

[31] N. J. Ward, The static analysis of safety critical software using MALPAS, in *Proc.* SAFECOMP'89 (Vienna, Austria, December 1989), 91-96.