Modeling Software Design Diversity – A Review

BEV LITTLEWOOD, PETER POPOV, and LORENZO STRIGINI

Centre for Software Reliability, City University

Design diversity has been used for many years now as a means of achieving a degree of fault tolerance in software-based systems. While there is clear evidence that the approach can be expected to deliver some increase in reliability compared to a single version, there is no agreement about the extent of this. More importantly, it remains difficult to evaluate exactly how reliable a particular diverse fault-tolerant system is. This difficulty arises because assumptions of independence of failures between different versions have been shown to be untenable: assessment of the actual level of dependence present is therefore needed, and this is difficult. In this tutorial, we survey the modeling issues here, with an emphasis upon the impact these have upon the problem of assessing the reliability of fault-tolerant systems. The intended audience is one of designers, assessors, and project managers with only a basic knowledge of probabilities, as well as reliability experts without detailed knowledge of software, who seek an introduction to the probabilistic issues in decisions about design diversity.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*fault tolerance; reliability, availability, and serviceability*; D.2 [**Software Engineering**]; J.7 [**Computer Applications**]: Computers in Other Systems

General Terms: Design, Reliability

Additional Key Words and Phrases: Control systems, functional diversity, multiple version programming, *N*-version software, protection systems, safety, software fault tolerance

1. INTRODUCTION AND BACKGROUND

1.1. The Need for Software Reliability

All systems need to be *sufficiently* reliable. Even for mass-market software, such as word-processors and spreadsheets, where the consequences of individual failures are usually not catastrophic, unreliability can have serious commercial implications for vendor and user. For safety-critical software, on the other hand, it is clearly vital that its unreliability is not greater than is needed for its contribution to the overall safety of a system.

There are two related issues here. In the first place there is the issue of achieving

Author's address: Centre for Software Reliability, City University, Northampton Square, London EC1V OHB, UK, e-mail: {bl; ptp; strigini}@csr.city.ac.uk.

This work was funded partially under the UK HSE Generic Nuclear Safety Research Programme (under the "DIverse Software PrOject" (DISPO)) and by the UK Engineering and Physical Sciences Research Council (EPSRC) (under the "Diversity In Safety Critical Software" (DISCS) project), and is published with permission.

the necessary reliability. Is the target reliability feasible? What software engineering techniques are appropriate to employ in its design and building? Secondly, there is the issue of assessing the reliability that has actually been achieved, to convince ourselves that it is "good enough."

Clearly, the difficulty of these tasks will depend upon the level of reliability that is required. This varies quite markedly from one application to another, and from one industry to another. 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 aircraft such as the Airbus A3XX and Boeing 777 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 [FAA 1985]. By contrast, safety systems—systems that are only called upon when some controlled system gets into a potentially dangerous state—often have relatively modest requirements; for example, the software-based primary protection system (PPS) for the Sizewell B nuclear reactor had a requirement of 10^{-4} probability of failure upon demand (*pfd*).[1]

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/monitoring systems give a failure rate of $4 \cdot 10^{-8}$ per hour for potentially safety-related functions [Laryd 1994]; an analysis [Shooman 1996] of FAA records (while pointing out the extreme difficulty of obtaining trustworthy data) tentatively estimated failure occurrence rates in avionics software to vary in the range 10^{-6} to 10^{-8} (very high reliability, but short of the 10^{-9} level) for systems in which failures prompted the issue of FAA "airworthiness directives," and a much lower bound for systems for which no such failures were reported. However, such after-the-event assessment of reliability is not the same as an assurance *prior to deployment* that a very high reliability has been achieved.

[1] A sensitivity study of the probabilistic risk assessment of the Sizewell B reactor later showed that a 10^{-3} *pfd* would still produce a tolerable risk and it is this latter figure that the UK Nuclear Installations Inspectorate has accepted.

1.2. Modeling Single-Version Software Reliability

1.2.1. The Software Failure Process. Before discussing the use of multiversion software in a fault-tolerant system, it is instructive to look briefly at the nature of the software failure process, and answer some of the common questions that are asked: Why does software fail? What are the mechanisms that underlie the software failure process? If software failures are "systematic," why do we still talk of reliability, using probability models?

We begin with the last of these, examining what is meant by the terms *random failure* for hardware and *systematic failure* for software. These do seem somewhat misleading, inasmuch as they appear to suggest that in the one case a probabilistic approach is inevitable, but that in the other we might be able to get away with completely deterministic arguments. 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, that is, 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 considered 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.[2]

However, our interest really centers upon the failure process: what we see when the system under study is used in its operational environment. In a real-time system, for example, we would have a well-defined time variable (not necessarily real clock time) and our interest would center upon the process of failures embedded in time. In this case 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 preassigned mission time. In a safety system, such as a reactor protection system, which is only required to respond to occasional demands from a wider system, we would be interested in the process of failed demands within the sequence of all demands. We might express our reliability requirement as a probability of failure upon demand. The important point is that the failure processes are not deterministic for either systematic or random faults, as we will show.

We use the terminology of software here for convenience, but it should be remembered that systematic failures also include those arising from certain design and construction faults in hardware. Indeed, the very success of the conventional physical hardware reliability theory is now revealing the importance of design faults to the overall reliability of complex systems even when these do not contain software. Our success in devising intelligent strategies to minimize 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.[3] These faults will have been resident in the software since their creation in the original design or in subsequent changes.

The software failure process, then, is a process in which faults are encountered as a result of execution on a succession of input readings. Consider, as an example, a nuclear plant's safety protection system, which must respond to the demands made upon it by a wider system (the physical reactor and its control system). The totality of all possible demands, the *demand space*, D (see Figure 1), is likely to be extremely large. Each point in this many-dimensional space can be thought of as completely characterizing a particular physical demand. This could be a vector of temperatures, pressures, flow rates, and the like, sampled at regular intervals by sensor scans (the period of time required to define a demand will influence the dimension of the vector).

Note that the notion of a succession of sensor readings, or "trajectory" is, in this example, incorporated into the definition of demand, so that a single point in the demand space describes the way in which a demand occurs and progresses through time. This allows us to regard a single point in the demand space as completely representing a particular demand, albeit at the price of making the space itself extremely complex. Readers should note that this model is only intended to be used at a conceptual level: it is unlikely that it would be possible to give a complete and detailed description of a demand space for most applications, but

[2] In practice, even design-caused failures may not occur in an obviously deterministic way. In software, it often happens that failures are difficult to reproduce because they depend on specific, difficult-to-observe conditions, like activities of other programs in the same computer. In hardware, some design faults will just make the system exceedingly vulnerable to some stressful condition (e.g., corrosion or electromagnetic interference). This fact only reinforces the need for a probabilistic approach to design faults.

[3] *Design*, in this context, means the whole process which produces the executable software: design, or software, faults are all defects in the executable software, caused, for example, by errors in defining the specifications for the software, by coding errors, by errors in compilation due to compiler defects, or by configuration errors, such as using the wrong release of some software modules. The definition excludes defects in the stored image of the software generated by physical causes, such as memory errors or communication noise.

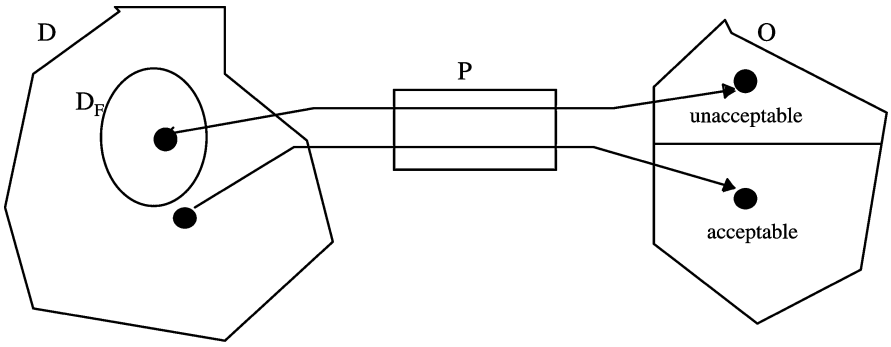

Fig. 1. Conceptual model of the software failure process. Program execution is a mapping from the set D, of all possible demands (sequences of input values), into the set of output sequences O. D_F represents the totality of all demands that the program P cannot execute correctly; they map into unacceptable output sequences.

this is not needed for our purposes here.[4] In Section 5.3, we discuss the difficulty in modeling the demands as explicit sequences of inputs.

When the software executes, demands are selected from the demand space and there is usually inherent uncertainty about this selection mechanism: we cannot predict with certainty what all future demands will be. This uncertainty can be represented, formally at least, by a probability distribution[5] over the space of all possible demands: we call this the *demand profile*. In the case of the protection system it might be reasonable to believe that the successive demands are selected independently according to this distribution, since successive demands are likely to be months apart and thus there is little chance of there being "memory" of a previous demand.

Some of the demands in the demand space are ones that the program can execute correctly; some are ones it cannot execute correctly: when one of the latter is encountered, we say that the software *failed*. We call the set of all failure-causing demands the *failure set* D_F. How we decide whether the result of a program execution is a failure is clearly not a trivial exercise, but is beyond the scope of the present discussion; typically it requires a specification of intended behavior that is sufficiently complete that a decision on acceptability of output can be made for all possible demands.

Since, as we have seen, there is uncertainty as to which demand will be selected on a particular occasion, there is also clearly uncertainty as to whether this demand will lie in D_F, that is, uncertainty as to whether there will be a failure. In other words, the process of failures of a program is an uncertain one: it is a *stochastic process*. It follows that all statements concerning reliability must take account of this inherent uncertainty: systematic failures are just as "random" as (conventionally defined) random failures. If, in the protection system example, we knew all the failure-prone demands that comprise D_F, and the probability associated with each such demand, the probability of failure upon demand could be predicted: it would be the sum of these probabilities. This is rarely a practical way of predicting

[4] In the previous literature, the *demand space* as defined here is usually called the *input space*. This has turned out to be confusing as the term *input* is also commonly used for individual "input variables" to a program or system. The set of values of the variables read in one sampling step, which is itself a multidimensional vector, does not by itself determine whether the software will produce a correct result or a failure, because the software's behavior is also determined by the values stored in its internal variables, which depend on the sequence of the previous input readings.

[5] Once again, it is unlikely in practice that this would be known in detail, but this is not needed for the conceptual model here to be useful.

reliability, of course, since information this extensive is unlikely to be available.

The reader will have seen that this discussion in terms of the demand space D and failure set D_F does not allow us to talk about particular faults. One way to think of a fault in this model is to ask what happens when a failure occurs, and the fault that caused the failure is removed by a (successful) change to the program. Clearly, this means that the offending demand can now be executed correctly by the program, and would not cause failure if it were selected again. In most cases it will not be the only demand so affected, and many points that were in D_F will now be executed correctly. In other words, D_F will have decreased in size. We can think of the subset of points in D_F that have changed from failure points to nonfailure points (*success points*) to be the "fault" that was removed. In the case of the protection system, the improvement in the *pfd* resulting from the fault's removal would simply be the probability of selecting a demand from this subset—the sum of the probabilities from the demand profile over this subset.

This interpretation of fault as a subset of D_F is not, of course, a semantic one. We cannot use our knowledge of it to say anything about the nature of the mistake that a human designer made, that has become embedded in the program. Conversely, it is also usually difficult, if not impossible, to use such semantic information, when it is available, to say anything useful in terms of the interpretation above. In particular, it is usually difficult to know what impact upon reliability there will be if we remove a particular fault, even when we have considerable information about its nature.[6]

1.2.2. Evaluation of Software Reliability from Failure Data. The purpose of the previous section was to show that the software failure process is indeed a random one, as is the case for failures in conventional hardware reliability studies. The underlying sources of the uncertainty in the two cases are, however, completely different.

Clearly, the conceptual model of the previous section is not very helpful when we need to estimate and predict the reliability of a particular program based upon observation of its actual failure data. Instead, research over the years has concentrated on building a statistical theory based upon the model.

The techniques for predicting future reliability from observed behavior can be divided into two categories, dealing with two different forms of the prediction problem:[7]

- *Steady-state* reliability estimation, considering the results of testing the version of the software that is to be deployed for operational use (as delivered); the theory underlying this prediction is much the same as that used in predicting the reliability of physical objects from sample testing.
- *Reliability growth*-based prediction (often called "reliability growth modeling"), considering the series of successive versions of the software that are created, tested, and corrected after tests discover faults, leading to the final version of the software that is to be evaluated. The data used in this case are the results (series of successful and of failed tests) of testing each successive version. Having observed a trend of (usually) increasing reliability, we can extrapolate this trend to predict current reliability and how it will change in the future.

Steady-state evaluation is the more straightforward procedure, and requires fewer assumptions. The behavior of the system in the past is seen as a sample from the space of its possible behaviors. The aspect of interest of this behavior, that is, the occurrence of failures, is governed by parameters (typically, a failure rate or a

[6] There are exceptions to this. If, for example, the fault takes the form of a particular function of the software simply not working, and we know how frequently the function is called upon in operational use, then this frequency tells us the increase in reliability that will result from removing the fault.

[7] A more complete introduction to the various reliability prediction methods, and the inference techniques they use, may be found in Littlewood and Strigini [1998].

probability of failure per demand) that can be estimated via standard inference techniques. Many projects, however, budget for little or no realistic testing of a completed design before its deployment, or in any case set reliability requirements higher than their budgeted amount of testing can confirm with the required confidence. Reliability growth-based prediction is then an appealing alternative, because it allows the assessor to use the evidence accumulated while the product was debugged rather than just evidence about its final version. 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.

In both cases, the success of prediction depends upon the observed failure process being similar to that which it is desired to predict: the techniques are essentially sophisticated forms of extrapolation. In particular, if we wish to predict the operational reliability of a program from failure data obtained during testing, it is necessary that the test case selection mechanism produce cases representative (statistically) of those that present themselves during operational use. This is not always easy, but there is a good understanding of appropriate techniques, as well as some experience of it being carried out in realistic industrial conditions, with the test-based predictions being validated by observation of later operational use [Littlewood and Strigini 1998; Dyer 1992; Musa 1993].

It is also worth emphasizing that, although we often speak loosely of the reliability of a software product, in fact we really mean the reliability of the product working in a particular environment, since the perceived reliability might vary considerably from one user or installation to another. It is not currently possible to test a program in one environment (i.e., with a given selection of test cases) and use the reliability growth modeling techniques to predict how reliable it will be in another. Essentially the problem will be to ensure that the testing environment is statistically identical (i.e., in the manner in which demands are selected) to the operational one.

By far the most extensive and successful work on software reliability assessment concerns reliability growth modelling. There is now a large body of literature together with some considerable experience of these techniques in industrial practice [Lyu 1996], and it is now often possible to obtain good predictions of the operational reliability of a program. These techniques might be first candidates for evaluating the reliability of a critical system, except for two obstacles: the aforementioned assumption that an observed statistical trend to increased reliability continues through the last fix, and the more general problem that obtaining assurance that a software product satisfies its reliability requirements via statistical techniques is only feasible, in practice, when the requirements are fairly modest. The reason is the law of diminishing returns in reliability growth.

Reliability growth models in their simplest form assume that when a failure occurs there is an attempt to identify and remove the design fault which caused the failure, whereupon the software is set running again, eventually to fail once again. The successive times of failure-free working are the input to statistical models, which use these data to estimate the current reliability of the program under study, and to predict how the reliability will change in the future.

Figure 2 shows an analysis of failure data from a system in operational use, for which software and hardware design changes were being introduced as a result of the failures. Here the current rate of occurrence of failures (ROCOF) is estimated at various times. The dotted line is fitted manually to give a visual impression of what seems to be a very clear law of diminishing returns. The level of reliability reached here is quite modest: about 10^{-2}

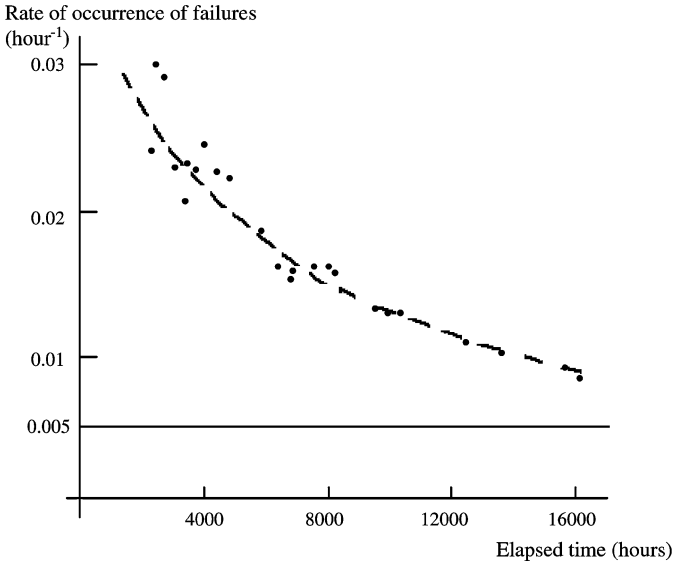

Fig. 2. Estimates of the rate of occurrence of failures for a system experiencing failures in operation due to software faults and hardware design faults. Here the points indicate recalculations of the estimate, performed periodically; there are many failures between any two successive points. The broken line here is fitted by eye.

failures per hour of operational use. More important, it is by no means obvious how the details of the future reliability growth of this system will look. For example, it is not clear to what the curve is asymptotic: will it eventually approach zero, or is there an irreducible level of residual unreliability reached when the effects of correct fault removal are balanced by those of new fault insertion? And what may this level be? Assuming, for instance, a requirement of a failure rate lower than 0.005 (the horizontal line in the figure) should one expect the curve shown to drop towards an asymptote that is below this level, or above it?

This empirical evidence of a law of diminishing returns for debugging software seems to be supported by most of the available evidence. There are convincing intuitive reasons for results of this kind.

A program starts life with a finite number of faults, and these are encountered randomly during operation. Different faults contribute differently to the overall unreliability of the program: some are larger than others. "Large" here means that the rate at which the fault would show itself (i.e., if we were not to remove it the first time we saw it) is large: different faults have different rates of occurrence. Adams [1984] shows a particularly dramatic example of this based on a large database of problem reports for some large IBM systems. The smallest faults he discovered each occurred only about once every 5,000 years. They accounted for 1/3 of uncovered faults.

During reliability growth we assume that a fix is carried out at each failure. Let us assume for simplicity that each fix attempt is successful. As debugging progresses, there will be a tendency for a fault with a larger rate to show itself before a fault with a smaller rate: more precisely, for any time t, the probability that fault A reveals itself during time t will be smaller than the probability that B reveals itself during t, if the rate of A is smaller than the rate of B. Informally, large faults get removed earlier than small ones. It follows that the improvements in the reliability of the program due to earlier fixes,

corresponding to faults which are likely to be larger, are greater than those due to later fixes.[8]

Thus, the law of diminishing returns shown in these examples is a result of two effects that reinforce each other. As debugging progresses and the program becomes more reliable, it becomes harder to find faults (because the rate at which the program is failing is becoming smaller), and the improvements to the reliability resulting from these fault removals are also becoming smaller and smaller.

In the discussion above, there has been an important implicit assumption that it is possible to fix a fault when it has been revealed during the test, and *to know that the fix is successful*. In fact, there has been no serious attempt to model the fault-fixing operation and most reliability growth models simply assume that fixes are perfect, or average out any short-term reversals to give the longer term trend.

The difficulty here is that the potential increase in unreliability due to a bad fix is unbounded. Even to have high confidence that the reliability was as high as it was immediately prior to the last failure, it would be necessary to have high confidence that no new fault had been introduced. There seem to be no good grounds to have such high confidence associated with a particular fix other than to exercise the software for a long time and never see a failure arise from the fix.

The conservative way forward in this case is to treat the program following a fix as if it were a new program, and thus take into account only the period of failure-free working that has been experienced since the last fix. This recasts the problem in terms of steady-state reliability assessment. Not surprisingly, the claims that can be made for the reliability of a system that has worked without failure are fairly modest for feasible periods of observation. Intuitively, observing a system to operate without failure over a short period of time would not give much confidence in correct operation over a much longer period. Using a rigorous inference procedure gives results such as: for a demand-based system such as a protection system, if we require 99% confidence that the *pfd* is no worse than 10^{-3}, we must see about 4,600 failure-free demands; for 99% confidence in 10^{-4}, the number increases to 46,000 failure-free demands. Such a test was completed for the Sizewell PPS (mentioned in Section 1.1). In the case of a continuously operating system, such as a control system, a 99% confidence in an MTTF of 10^4 hours (1.14 years) would require approximately 46,000 hours of failure-free testing; to raise the confidence bound on the MTTF to 10^5 hours, the testing duration must also increase to approximately 460,000 hours. In summary, high confidence in long failure-free operation in the future requires observing much longer failure-free operation under test. If this amount of test effort is not feasible, only much lower confidence can be obtained. Even if we have other sources of confidence in the software, we still find that observing correct behavior over a short period of time adds very little to any confidence we may have in reliability over long future periods [Littlewood and Strigini 1993].

2. WHY DESIGN DIVERSITY?

2.1. Motivation and Principles

In the light of the rather strict limitations to the levels of software reliability that can be claimed from observation of operational behavior of a single version program, fault tolerance via design diversity has been suggested as a way forward both for achieving higher levels of reliability and for assisting in its assessment.

The intuitive rationale behind the use of design diversity is simply the age-old human belief that "two heads are better than one." For example, we are more likely to trust our answer to some complex arithmetic calculation if a colleague has arrived

[8] It will now be clear that the assumption of successful fixes is not essential for this argument. Even if some fixes are partially or totally ineffective, the reliability improvement due to a fix can be at most equal to the rate of manifestation of the fault that is fixed, and this tends to decrease from earlier to later fixes.

independently at the same answer. In this regard, Charles Babbage [1974] was probably the first person to advocate using two computers, although by computer he meant a person.

People building hardware systems have known for a very long time that the reliability of a system can be increased if redundancy can be built into its design. Thus, when a component fails, if there is another component waiting to take over its task, the failure can be masked. Indeed, if we were able to claim that components failed independently of each other, we might claim to make arbitrarily reliable systems from arbitrarily unreliable components. In practice, though, such statements are largely mathematical curiosities, since complete independence rarely, if ever, occurs in practice (and complex redundant structures bring their own, novel, forms of unreliability). We show later that it is this issue of dependence of failures that makes modeling of software fault tolerance particularly difficult.

Of course, simply replicating a component (hardware or software) with one or more identical copies of itself will provide no guarantee against the effect of design faults, since these will themselves simply be replicated. If all copies are exposed to the same demands, whenever a demand triggers a design fault all versions will fail together. The natural defense against design faults in a component is thus to add different code, which may not be subject to the same faults.[9]

In its simplest form, design diversity involves the "independent"[10] creation of two or more versions of a program, which are all executed on each input reading so that an adjudication mechanism can produce a "best" single output. Here the versions may be developed from the same specification, or the higher-level (and usually more informal) engineering requirements may be used to produce diverse formal specifications. Typically, the teams building the versions work "independently" of one another, without means of direct communication (indirect communication may still occur: for example, one team may discover faults in the common specification, causing the project managers to issue a correction to all teams). The teams may be allowed complete freedom of choice in the methods and tools used, or they may have these imposed upon them in order to "force" diversity (e.g., different programming languages). In the former case, the hope is that identical mistakes will be avoided by the natural, "random" variation among people and their circumstances; in the latter, the same purpose is pursued by intentionally varying the circumstances and constraints under which the people work to solve the given problem.

Of course, this is a somewhat simplistic view of diversity, and not all systems that use design diversity do so at this high

[9] An alternative, partial defense is to introduce differences between the demands to identical copies of the component ("data diversity"); implicitly, via loosely coupled execution of the software copies and sampling of their sensor inputs, or explicitly, by seeding small differences between the inputs to the components. Loosely coupled replication has been widely adopted, even without explicit consideration for the design faults. In records of operation of Tandem fault-tolerant systems (with two copies of the software running on loosely coupled computers), for instance, it tolerated [Lee and Iyer 1995] 82% of the software-caused failures: many software failures only happen in specific states of the operating system and application processes, which do not occur identically on the two machines. For intentionally seeded discrepancies between inputs, Ammann and Knight [1988] experimented with the software versions used in the Knight-Leveson experiment: they found that, on a failure-causing demand, retry with a slightly different demand would only cause a failure with a probability that varied, for different faults, from 0 to 99%. On the same principle, it has been shown that unreliable software may be made more reliable by frequent restarts ("software rejuvenation" [Huang et al. 1995]) that reset state variables of the system, purging them of erroneous values or other unusual, untested-for conditions they may have reached.

[10] We use quotes here, and later in this paragraph, because of the profligate way in which words such as "independence" and "independent" are used in writing about diversity and fault tolerance. Strictly, the only use of the terms which has a formal definition concerns statistical independence, for example here between the failure processes of two or more versions. We do not persist in this use of quotes in the rest of the article because of their stylistic inelegance, but we hope that the different meanings are evident from the context.

level. Diversity can be used at lower levels in the system architecture, for example, to provide protection against failures of particularly important functions, and in a variety of forms. Designers may choose to adjudge a correct result by some form of comparison or voting, or by using self-checks or acceptance tests to detect and exclude incorrect results [Di Giandomenico and Strigini 1990; Blough and Sullivan 1990]. A correct state of an executing software version can be recovered after failures by forward recovery (by adjudicating between the alternative values available) or by rollback and retry; diverse software versions may be allocated to processors, and scheduled to execute, according to various alternative schemes, adapted to the kind of hardware redundancy present. The hardware processors themselves will often be diverse, for protection against the design faults in the processors, which are known to be common [Lyu 1995; Voges 1988; Laprie et al. 1990]. Widely known, simple fault-tolerant schemes are: pure *N-version* software, with multiple versions produced as we outlined above, and executed on the redundant processors of an *N*-modular redundant system; *recovery blocks*, in which one version is executed at a time, its failures are detected by an acceptance test, and recovered via roll-back and retry with a different version of the software; and *N-self checking* software, in which, for instance, version pairs are used as self-checking components, which self-exclude when a discrepancy between the pair is detected: for instance, two such pairs form a redundant system able to tolerate the failure of any one of the four versions. More generally, some form of diversity is used against design faults in most well-built software, in the form of defensive programming, exception handling and so on. These defenses are often dispersed throughout the code of a program, but they may also form a clearly separate subsystem, which monitors the behavior of the main software, for instance, to guarantee that the commands to a controlled system remain within an assigned safe envelope of operation [Kantz and Koza 1995]. These methods are different from the writing of multiple diverse versions of the software, but the main question about their effectiveness is the same: how likely is it for the defensive code to fail in coincidence with the failures against which it is intended to be a defense?

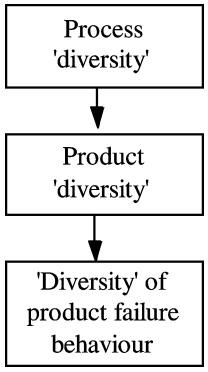

Fig. 3. Different types of diversity at different stages of the software design and development process.

For simplicity of exposition in this article, we generally restrict ourselves to the simple case of multiple- (usually two-) version software, since our main concern is with issues of assessment of reliability, rather than architectural issues.

Figure 3 shows the way in which it is generally believed, in an informal way, that design diversity works. At the top level, there is process diversity, that is, diversity in the design practices, in the personnel, and so on, involved in the production of the versions. This diversity—people doing things differently—results in the versions themselves being different from each other, and in particular, it is hoped, it is unlikely that faults in one version will be identical to those in another. These differences between versions in turn affect the chance that, if one version fails on a particular demand, another version will also fail: it is this diversity of failure behavior that is the goal we seek. We would like statistical independence between the version failure processes or, even better, a negative dependence, so that the situations in which one version is especially likely to fail will be ones in which the other is especially unlikely to fail.

Claims for the reliability of a software-based system that utilizes design diversity (perhaps forming part of the safety case of a wider system) will involve evidence from all three stages shown in Figure 3, and might feed into a safety case argument. Positive evidence from each of these stages would tend to make us have more confidence in the system but we would clearly need to qualify and quantify this higher confidence on the basis of the strength of the evidence and the stage it concerns.

2.2. Acceptance

We turn now briefly to the degree of industrial adoption of diversity. The very success of techniques using redundancy to protect against hardware failures has resulted in a greater proportion of failures being due to design defects. At the same time, in most industries there has been an increase in system complexity as designers take advantage of the extensive functionality that can be provided by software, so the risk of human errors resulting in design faults has increased. Also, software-based systems are being given increasingly critical tasks, for instance, with software-based safety protection systems substituting hardwired ones. Design diversity was thus adopted in some industrial sectors (aerospace and rail transportation) as software was beginning to be used for safety-critical functions. Examples include the Airbus A320/30/40 aircraft [Traverse 1988; Briere and Traverse 1993] and various railway signaling and control systems.[11] The adoption of diversity has been limited, though, by doubts about its costs and effectiveness, which we discuss in the next section.

Software diversity is also present as a side-effect in many systems using *functional* diversity, the widespread system design approach whereby critical system functions are provided by multiple subsystems that differ as much as possible in their principles of action, inputs, and implementation technology: for instance, giving an aircraft different instruments, based on different physical principles, for estimating speed or altitude.

As for software design diversity without differentiation of functions, the attitudes of industry and regulators vary not only between industrial sectors but also within the same sector. The accepted guidelines for the civil aviation industry, RTCA/EuroCAE [1992] do not prescribe software diversity; they allow a company to claim diversity as one alternative to some other, standard assurance practices, but require the company to demonstrate the specific benefits claimed from its use of diversity. Other standards (e.g., MoD [1997; 1996]) also consider diversity, but generally with the same attitude of leaving to the developer the burden of demonstrating its advantage. As we said, the Airbus A320/330/340 families of aircraft use software diversity (although their airworthiness certification did not rely on this: diversity was just an additional factor to increase confidence in aircraft that depended on software as no other civil aircraft had before). Boeing, on the other hand, decided against software diversity for its own 777 aircraft, on the grounds that it would require restrictions to communication between software and system engineers, which in turn is an important defense against requirement errors [Yeh 1998]. Boeing (like Airbus) instead used hardware diversity among the redundant processors.

3. DOES DESIGN DIVERSITY WORK?

Evidence concerning the effectiveness of design diversity falls into three main types:

- operational experience of its application in real industrial systems;
- controlled experimental studies; and
- mathematical models of the failure processes of diverse versions.

Many industrial and research experiences are reported in Lyu [1995] and Voges [1988], but relatively little data have been published. On the positive side,

[11] See, for example, Hagelin [1988], Mongardi [1993], Turner et al. [1987], Kantz and Koza [1995], and Lindenberg [1993].

several safety-critical systems have been implemented using software fault tolerance based on design diversity, and there have been no reports of catastrophic failure attributable to software design faults.

It seems reasonable to believe that diversity contributed to reliability and safety in these applications, although the evidence is insufficient to decide how much it helped. Disagreements between versions have indeed happened. A disagreement may mean that one of the versions was in error due to a design fault, and thus diversity actually acted to prevent a possible system failure. These "vote-outs" in themselves do not necessarily indicate a successfully tolerated design-caused failure: they might also be the effect of transient hardware faults (which could be tolerated by simple redundancy without the extra cost of diversity) or even occasional spurious disagreements between correct versions. The manufacturers have reported that some software faults were found, but they were of a nonthreatening nature. So, the evidence from operation of these systems is a weak indication of usefulness. However, even if these specific systems had been free of design faults, this would not make the decision to apply diversity an unreasonable one; software development processes produce very variable results, and diversity would act as insurance against the risk of a single version system being an unusually unreliable product.

On the other hand, there are insufficient data, at least in the public domain, to be able to say whether the use of design diversity in these examples resulted in more reliable systems than could have been achieved for the same effort by other means. The outstanding issues are as follows.

(1) For applications with extreme reliability requirements (as in railway and avionics systems), can diversity, added to "complete" exploitation of the other techniques available, improve the reliability that can be achieved, albeit at added cost? The a priori answer seems to be yes, provided that the added architectural complexity does not offset the gain due to diversity; but strong a posteriori evidence would be very difficult to obtain, as even single versions would be very reliable.

(2) Otherwise, the question becomes one of cost-effectiveness: to what extent (or in what circumstances) is design diversity a more effective way of achieving a particular level of reliability in a software-based system?

The costs of diversity have several components. Of course each software version must be developed and verified. Activities such as coding are fully replicated for each version produced, multiplying costs accordingly. Other costs may increase, but not linearly with the number of versions. For example, the possibility of testing the multiple versions back to back may reduce the cost of verification (compared to verifying N versions separately). Requirements need to be specified only once, but the need to avoid ambiguities leading to discrepancies between the versions may make the requirement phase more expensive (although possibly higher quality) than for a single version. A redundant architecture is clearly more complex to design than a nonredundant one, and even for a system that would employ hardware redundancy in any case, diversity requires additional attention in designing, for instance, the adjudication functions. Last, the development of multiple versions has greater organizational costs than that of one version, in terms of coordination effort, cost of delays, and so on. The cost of diversity has been discussed for example, in Migneault [1982], Laprie et al. [1990], and Voges [1994].

Some information on this cost-effectiveness issue (albeit incomplete) comes from experiments that have been conducted under controlled conditions. In Anderson et al. [1985], it is reported that a two-version recovery block system masked about 70% of the failures that would have taken place in the single channel of the primary version. That is, the two-version system is less than half as unreliable as the single version. We might be tempted

to assume that a two-version system costs twice as much to develop as a single version, and conclude that there is a modest advantage to be gained from the use of diversity. This could be misleading, though, since the cost of improving the reliability of a single version may not increase linearly: we have seen earlier, for example, that for testing, at least, there is a severe law of diminishing returns. So, given that a two-version system achieves a given reliability, it may be the case that producing a single version to achieve the same reliability by itself would cost more than the two-version system. Besides, in some applications (including the one in this experiment) the second version need not be as complete in its functionality as the first one. These observations support the view that design diversity is effective.

In the above experiment, there were only two versions because the system was built to commercial standards, using expensive industrial designers and programmers. Similar constraints apply to several experiments in the nuclear field.[12] Other experiments have developed sufficient numbers of versions to permit statistical analysis, but often at the price of using students as programmers, and involving toy programs.

Several experiments were conducted in the US during the 1980s under NASA sponsorship. One of the best-known of these was carried out by John Knight and Nancy Leveson [1986] at the Universities of Virginia and California, respectively. The original intention of the authors was to carry out a statistical test of the hypothesis that independently developed versions failed independently. The experimental results allowed the authors to reject this hypothesis resoundingly: there was overwhelming evidence that simultaneous version failures were more likely than would be the case if they were failing independently. This negative result, and similar results from later experiments, have often been cited as a reason for not using design diversity for software-based systems, particularly in the US.

In fact, this negative result is only part of the story: while rejecting the hypothesis of independence, the authors also investigated whether there were nevertheless reliability benefits from the software diversity. The experiment involved developing 27 versions and subjecting them to 1,000,000 test cases against an oracle version that was presumed correct. On each test case, a vector of 27 dimensions recorded the result—correct or incorrect—of each version. The authors were thus able to calculate the hypothetical reliabilities of fault-tolerant architectures comprising different versions. For example, they examined all two-out-of-three systems that could be constructed, and found that the average reliability among these was an order of magnitude better than the average reliability of the 27 single versions.

This is, again, quite a positive result, but a word of warning is appropriate. The results relate to averages: there was great variation between individual version reliabilities, and between the reliabilities of two-out-of-three systems. In fact some of the two-out-of-three systems were considerably less reliable than some of the single versions. Thus even if we could be sure that this result would be reproduced on different, more realistic, problems, it would not allow us to make strong claims for the reliability of a particular two-out-of-three system. This difference between what we might expect on average, and what we achieve in a particular instance, will turn out to be a key problem when we come to look at the detailed mathematical models of diversity. In practice, of course (particularly for safety-critical systems) we want to make trustworthy claims for a particular system.

All controlled experiments we are aware of produced similar results: multiple-version systems were, on average, more reliable than individual versions, and sometimes much more so. Such experiments cannot tell how much diversity would improve reliability in a new industrial project, whether this improvement could

[12] See, for example, Voges and Gmeiner [1979], Bishop [1988], Bishop and Pullen [1988], Smith et al. [1991], and Kersken and Saglietti [1992].

be achieved by other means, and which methods would be more cost-effective. However, costs are directly observable after each development. The first issue confronting developers and regulators is how much of a reliability gain can be expected from diversity, which is the issue we examine in the following sections.

4. PROBABILITY MODELS FOR CONCEPTUAL UNDERSTANDING OF SOFTWARE DIVERSITY

In this section, we outline the two probabilistic models developed in the 1980s that started explaining the observation of positive correlation between failures of diverse versions. Mathematical details for both can be found in Littlewood and Miller [1989].

One explanation for the fact that people tend to make similar mistakes in certain circumstances is that some problems are intrinsically harder than others; that is, there may be some parts of a programming task that most people will find difficult. This intuition is at the heart of the first probability model that attempts to capture the nature of failure dependency [Eckhardt and Lee 1985]. This elegant and influential model is based on a notion of "variation of difficulty" over the demand space. It is shown that the greater this variation, the greater the dependence in failure behavior between versions (and thus the less benefit in a fault-tolerant system). In particular, it is shown that even versions that are truly independently developed (and there is a precise meaning given to this in the model) will fail dependently.

Most of the theoretical results that we describe refer to the simplest form of diverse-redundant architecture: a two-version system, with perfect adjudication (one-out-of-two, diverse system). This very simple scheme is actually representative of important practical applications, like a plant protection system, in which two versions run on completely separate and noncommunicating hardware channels (sensors, computers, and actuators), and either version is able to order a shutdown

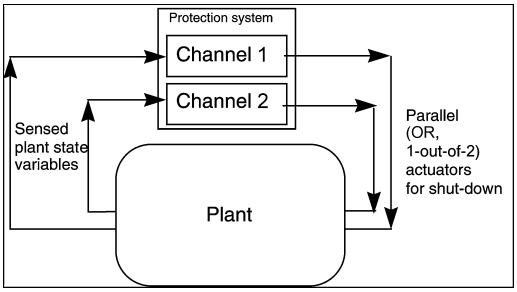

Fig. 4. Dual-channel protection system, stylized view.

action no matter what the other does. This is depicted in Figure 4. For this system, we study the probability of the event that both versions fail on the same demand. This is the basic problem in predicting the reliability of any diverse-redundant system. More complex systems may add details to the modeling problem (probabilities of common failure of subsets of the versions, or probability that although two versions fail they produce different outputs so that the failure may be detected), but none can be usefully addressed without addressing this basic problem first.

4.1. The Eckhardt and Lee (EL) Model

There are two basic sources of uncertainty or randomness, in the EL model. First, there is the random selection of demands from the space of all demands. This selection is controlled by a probability distribution over the demand space, which can be thought of as characterizing the demand profile.

Second, there is uncertainty associated with the creation of a program. The idea here is that there is a population of all programs that could ever be written, and the act of writing a program to solve a particular problem is a selection from this population via a probability distribution that characterizes the problem. This is an unusual idea but, as we will show, allows a quite intuitive development of the model. Clearly, we would never know this distribution in reality; nor would we be able to describe the space of all programs. Some programs would have a zero chance

of selection (e.g., completely inappropriate ones that do not address the problem); some programs would have a positive chance of selection (including, it is to be expected, "correct" ones, as well as programs that address the right functionality but contain faults). The key variable in the model is then the difficulty function $\theta(x)$, defined to be the probability that a program chosen at random will fail on a particular demand x. Here the program is chosen via the probability distribution over all programs, above. That is, if we were to select very many programs independently in this way, $\theta(x)$ would be the proportion of these that failed when presented with demand x. This seems a natural definition of the intuitive notion of difficulty: the more difficult a demand, the greater we would believe the chance that an unknown program will fail.

The important point here is that difficulty varies across the demand space. For a randomly selected demand, then, the difficulty is a random variable. The unreliability of a randomly selected program is then simply

$$P(\text{randomly selected program fails on randomly selected input}) = E_X(\theta(X)), \tag{1}$$

where we use the uppercase X to indicate that this is a random variable. $E_X(A)$ denotes the expected value of the random variable A.

Consider now the independent development of two programs π_1, π_2; in EL this is the *independent random selection* of π_1, π_2. It is easy to show that for any given x these two (randomly chosen) programs fail independently:

$$\begin{aligned} &P(\pi_1, \pi_2 \text{ both fail} \mid \text{input is } x) \\ &\quad = P(\pi_1 \text{ fails} \mid \text{input is } x) \\ &\qquad \times P(\pi_2 \text{ fails} \mid \text{input is } x) \\ &\quad = [\theta(x)]^2 \end{aligned} \tag{2}$$

or

$$\begin{aligned} &P(\pi_2 \text{ fails on } x \mid \pi_1 \text{ fails on } x) \\ &\quad = P(\pi_2 \text{ fails on } x) \\ &\quad = \theta(x), \end{aligned} \tag{3}$$

where the '|' symbol means "conditional on."

There is thus *conditional* independence in their failure behavior: they are independent for any *given* x ((2) and (3) are true for all x). The important achievement of EL is to show that even in the event the versions are developed independently so that there is this conditional independence, the versions will nevertheless fail *dependently* in the important case of a randomly selected (unknown) demand. This can be seen as follows. Consider the probability that a randomly selected pair of programs both fail on a randomly selected demand, denoted by X; this probability is

$$\begin{aligned} &\sum_x P(X = x) P(\pi_1, \pi_2 \text{ both fail} \mid \text{input is } x) \\ &\quad = E_X([\theta(X)]^2) \\ &\quad = (E_X[\theta(X)])^2 + Var_X[\theta(X)]. \end{aligned} \tag{4}$$

By $Var_X(A)$ we denote the variance of the random variable A.

It is expression (4) which represents the "unreliability" of a randomly chosen one-out-of-two system. Here the first term on the right is the naive result we would get from an incorrect assumption of independence, simply the product of the version probabilities of failure on a randomly selected demand. The second term $Var_X(A)$, is always nonnegative. Thus incorrectly assuming independence would underestimate the unreliability of a randomly chosen one-out-of-two system (the probability of both versions failing) by an amount given by the variance of the difficulty function over the demand space. The more the difficulty varies between demands, the worse the problem.

EXAMPLE

A simple contrived example may make clearer the ideas used in the EL model. Assume that the demand space of a program consists of only five different demands $D = \{x_1, x_2, x_3, x_4, x_5\}$, with the demand profile represented by the probability distribution Q given in Table I.

Table I. Illustration of How the EL Model Works. The Notations Used are: Demands $\{x_i\}$, Demand Profile $\{Q(x_i)\}$, Population of Versions $\{\pi_i\}$, Probabilities of Versions $\{P(\pi_i)\}$.

		Demands				
		x_1	x_2	x_3	x_4	x_5
	P(π_i)	$Q(x_1) = 0.99$	$Q(x_2) = 0.001$	$Q(x_3) = 0.004$	$Q(x_4) = 0.0045$	$Q(x_5) = 0.0005$
π_1	$P(\pi_1) = 0.1$	0	1	0	1	0
π_2	$P(\pi_2) = 0.2$	0	0	1	1	0
π_3	$P(\pi_3) = 0.4$	0	0	1	1	0
π_4	$P(\pi_4) = 0.3$	0	0	0	0	1
	$\theta(x)$	0	0.1	0.6	0.7	0.3

Assume further that the population of all possible programs is $\{\pi_1,\ \pi_2,\ \pi_3,\ \pi_4\}$. These are written to the same specification and if correct they would produce identical results on each of the five demands. Denote by $P(\pi_i)$ the probability that if asked to write software to the given specification, development teams would create version π_i. The important point here is that some versions will be more likely to be created than others. In reality, we might find that some versions are impossible, in which case $P(\pi_i) = 0$. Since $P(\pi_i)$ is a probability, $\sum_{i=1}^{4} P(\pi_i) = 1$. We can think of $P(\pi_i)$ as characterizing the process of version development. If we change the process, the probabilities $P(\pi_i)$ will change.

When the versions are executed, each will either succeed or fail on each demand. Table I shows how the four programs behave on the five demands: here 1 denotes a failure; 0 denotes a success.

The value of the difficulty function $\theta(x)$ on demand x will be the weighted sum of the 0s and 1s of versions in the column for that demand, the weights being the corresponding probabilities of versions $P(\pi_i)$. Note that since these weights characterize the development process, the difficulty of a demand depends on the process used to develop versions. This fits in with intuition: we might expect the failure-proneness of demands to vary according to the kind of software development used.

In this example, demand x_1 is correctly processed by all versions and therefore its difficulty function $\theta(x_1) = 0$. The values of the difficulty function on the other demands vary between 0.1 and 0.7.

The probability of failure of a randomly selected program on a randomly selected demand is the expected value $E[\theta(X)] = \sum_{i=1}^{5} Q(x_i)\theta(x_i) = 0.0058$.

For the randomly selected one-out-of-two system, according to (4), the probability of failure on a randomly selected demand is:

$$E[(\theta(X))^2] = \sum_{i=1}^{5} Q(x_i)\,\theta^2(x_i) = 0.0037.$$

In contrast, the naïve assumption of independent failures of both channels would give a very low probability of system failure ($E[\theta(X)])^2 = (0.0058)^2 = 0.0000336$. In other words, the one-out-of-two system is more reliable than a single-version system but substantially worse than would be obtained by assuming (incorrectly) that the versions fail independently.

4.2. Discussion and Intuitive Rationale

The Eckhardt and Lee [1985] result is an important one because it is the first serious attempt to model in a formal way the meaning of "independent development" of software versions. The early practitioners, and their academic counterparts, had a quite informal understanding of what was meant by "building two versions independently." "Independence" here seemed to be essentially about process: it meant mainly that the teams did not communicate with one another while building their respective versions. Similarly, there was little thought given to how failure independence might arise even if the versions were truly built independently. Rather it seemed to be assumed that if the different version development processes could be

controlled so that they could be claimed to be "independent" (a task that was admitted to be difficult), a claim for statistical independence of failures could reasonably follow.[13]

In the previous paragraph, we have used "independent" (in quotes) to refer to the imprecisely defined achievement of naturally occurring difference in the development processes, reserving *independent* (without quotes) to mean *statistically independent* (failure processes). The achievement of the EL model is that it gives a statistical meaning to the former, as well as the latter. Thus (2) and (3) above can be regarded as meaning that the "independent developments" have succeeded in building independent versions, in the sense that, for every demand, knowing whether one version fails or not does not tell us anything about whether the other version will fail.

This formal interpretation of what is meant by independent versions is, in fact, quite a strong one; advocates of design diversity might regard it as an ideal, but unrealizable, goal. Eckhardt and Lee [1985] show, however, that even this strong interpretation of version independence falls short of what is needed to claim the unconditional independence that is the real goal. Even if the strong *conditional* independence of (2) and (3) is true, the versions will still fail in a way that is positively correlated: a one-out-of-two system will be less reliable than it would be if the versions really did fail independently.

The key to the model lies in the variation of difficulty across the demand space. Thus, when a demand is selected at random, the corresponding difficulty must be treated as a random variable; seeing a version fail tells us something about this random variable, and so changes our distribution for it. In fact, it can be shown [Littlewood and Miller 1989] that the distributions for $\theta(X)$ before and after seeing a version failing are *stochastically ordered*:

$$\theta(X)|\pi_1 \text{ fails} \underset{\text{stoch}}{>} \theta(X). \tag{5}$$

The main EL result arises from a quite subtle interplay between conditional and unconditional independence. Conditionally, that is, if we know the demand x, the versions fail independently; thus knowing that π_1 failed does not change our belief that π_2 will fail. For a new (randomly chosen) unknown demand, however, seeing π_1 fail makes us believe that it is probably a difficult demand (this is the essence of result (5)) and thus increases the chance that π_2 would also fail. That is, for a randomly chosen X [Littlewood and Miller 1989]:

$$\begin{aligned} &P(\pi_2 \text{ fails} \mid \pi_1 \text{ fails}) \\ &\quad = \frac{P(\pi_1, \pi_2 \textit{ both fail})}{P(\pi_1 \textit{ fails})} \\ &\quad = \frac{(E_x[\theta(X)])^2 + Var_x[\theta(X)]}{E_x[\theta(X)]} \\ &\quad = (E_x[\theta(X)]) + \frac{Var_x[\theta(X)]}{E_x[\theta(X)]} \\ &\quad = P(\pi_2 \textit{ fails}) + \frac{Var_x[\theta(X)]}{E_x[\theta(X)]} \\ &\quad \geq P(\pi_2 \text{ fails}). \end{aligned} \tag{6}$$

There is equality here if and only if the variance is zero; that is, $\theta(x) \equiv \theta$ identically for all x. In other words, independence of failures of versions is only possible if there is no variation in difficulty. This seems so unlikely in real problems that it is fair to claim that there will always be positive correlation, and thus system reliability can always be expected to be lower than it would be if there were independence.

The EL model does not, unfortunately, help us in estimating the reliabilities of particular fault-tolerant diverse systems. It involves parameters that are unlikely to be estimable in practice. We are unlikely to be able to estimate $\theta(x)$ for any particular x, since this would require us to have a lot of independently developed programs that

[13] For an interesting discussion of the controversy that surrounded some of these early claims, see Knight and Leveson [1990] and the sources quoted there.

we could use to execute x. Thus the key variance term in (6) will not be estimable for a new development, although Nicola and Goyal [1990] show ways of estimating it given a sample of many developed versions of a program obtained in experiments.

The main result of EL is a negative one. It tells us that claims for independence of failures even from diverse software versions cannot be justified. While it should be emphasized that the results do not say that diversity is ineffective as a means of *achieving* high reliability, it means that reliability *evaluation* does not benefit from the fact of the presence of diversity; we cannot simply argue directly that two 10^{-3} versions will give us a 10^{-6} two-version system, so we must actually measure the reliability of this system.

In the next section, we describe the Littlewood and Miller model, which gives a theoretical way out of this impasse, albeit at the expense of difficulties of other kinds.

4.3. The Littlewood and Miller (LM) Model

In EL, independent program development is represented as the independent selection of programs from the population of all programs that could be written. This is the situation in which any differences between the versions will arise naturally from the fact of the teams being different and their not communicating with one another.

Instead of merely allowing diversity to arise willy-nilly in this way, another approach is to force diversity in the development processes of the different versions. Thus it might be insisted that the different teams use different programming languages, different testing regimes, and so on. Such forced diversity has been used in real industrial practice (e.g., the Airbus A320 flight control software [Briere and Traverse 1993] and in experiments (e.g., the DARTS project [Smith et al. 1991]).

The LM model generalizes the EL model to take account of forced diversity by defining different distributions over the population of all programs. The set of constraints imposed on the development of a version is called in Littlewood and Miller [1989] a *methodology*, which corresponds to a specific distribution of programs. Thus a program will have a different probability of selection (i.e., being developed) under methodology A from that under methodology B. In practice, some programs will be impossible under one methodology, and will thus have zero probability. The effect of there being these different distributions over the programs is that there are different difficulty functions induced over the demand space. Thus the probability of a program randomly chosen from methodology A failing on demand x is denoted by $\theta_A(x)$, with a similar interpretation for $\theta_B(x)$. The probability of a randomly selected A program failing on a randomly selected demand is $E_x(\theta_A(X))$, in a similar notation as before.

Once again, we could imagine, in an idealized experiment, estimating $\theta_A(x)$ by independently selecting many programs from methodology A, executing each on demand x, and calculating the proportion that fail.

Clearly, the EL model is a special case of LM when $\theta_A(x) = \theta_B(x)$ for all x. Interest centers upon cases where these two difficulty functions are different. Informally, we would like to have difficulty functions such that what is difficult for A is not difficult for B, and vice versa; that is, for those x for which $\theta_A(x)$ is large, $\theta_B(x)$ tends to be small, and vice versa. This would be the case if the difficulty functions were negatively correlated.

Consider now the independent development (i.e., selection) of a program using methodology A and a program using methodology B, π_A and π_B. Once again, it is easy to show that, for any particular demand x, these two programs will fail independently:

$$P(\pi_A, \pi_B \text{ fail on } x) = \theta_A(x)\theta_B(x) \qquad (7)$$

or, putting it a different way

$$P(\pi_B \text{ fails on } x \mid \pi_A \text{ fails on } x) = P(\pi_B \text{ fails on } x) = \theta_B(x). \qquad (8)$$

This is just a generalization of the conditional failure independence in the EL model. Once again, however, we are interested in the unconditional probability of a randomly selected pair of programs, one from A and one from B, both failing on a *randomly* selected demand. This is

$$E_x(\theta_A(X)\theta_B(X)) = E_x[\theta_A(X)]E_x[\theta_B(X)] + Cov(\theta_A(X))\theta_B(X)), \quad (9)$$

where $Cov(A, B)$ denotes the covariance of two random variables A and B.

As before, in (4), the incorrect, naive independent failures result is the first term on the right-hand side: it is merely the product of the A and B probabilities of failure. The interesting difference between this result and that of EL, however, is that since the covariance term on the right can be positive or negative, it is no longer certain that the probability of failure of both randomly selected versions will be greater than in the independence case, as was in (4).

4.4. Discussion and Implications of the Model

The basic intuition that underpins the LM model is simply the notion that what you find difficult, I may find easy (or at least easier), and vice versa.

The possibility of negative correlation between two difficulty functions means that the reliability of a one-out-of-two system could be greater even than it would be under an assumption of independence (given the same version reliabilities in the two cases). Whether this result has practical usefulness remains moot. It may turn out to be little more than an interesting mathematical curiosity; certainly there seems no way that such an assumption of negative correlation could be justified currently in a real application. If we could assume negative correlation (or justify it via some other arguments, such as analysis of the built versions), we would be in the enviable position of being able to use the independence-based estimate of the system reliability as a conservative bound on the real reliability.[14]

Even without the very strong assumption of negative correlation, however, the LM model goes some way to rescue design diversity from the pessimistic conclusions that arise from the EL model. Specifically, it seems reasonable to believe that the difficulty functions A and B will always be different. There will always be some differences between the programming teams, or the methods that they use, that are significant in determining the nature of the errors that they make. Even if, as seems likely, the difficulty functions are positively correlated, the expected reliability of a one-out-of-two system will always be greater under the LM model assumptions than under those of EL (keeping the version reliabilities fixed between the two cases). In this sense, EL can be seen as the most extremely pessimistic case within the more general LM model, and it is reasonable to think that it is unattainable.

In Littlewood and Miller [1989], the authors go further than this and provide some additional results for forced diversity. For example, they prove that for one-out-of-n systems, when fewer than n methodologies are available, all things being equal, the best design is the one that uses all the methodologies and spreads them as evenly as possible; for example, in the case where $n = 5$ and there are three methodologies available, it is better to build a $AABBC$ design than a $AAABC$ one.

It might be thought that this result is intuitively appealing and unsurprising;

[14] Littlewood and Miller [1989] compute the correlations from data obtained in the Knight and Leveson experiment, and data obtained from an error-seeding experiment by Knight and Amman [1985]. In the first instance, the correlation is positive, in the second, it is negative. This latter case is, in fact, rather a contrived one, but it does show that negative correlation is possible. However, to establish its presence here requires samples from the method A and method B programs (13 method A programs and 29 method B programs); such data will never be available in realistic nonexperimental situations.

after all, it essentially only says that diversity is a good thing. However, care needs to be taken here since similarly obvious results turn out to be false. For example, it can be shown that diversity is not necessarily a good thing in the case of two-out-of-three systems (or, in general, in $(n + 1)$-out-of-$(2n + 1)$ systems). One can build examples of triples of methodologies, producing versions with the same probability of failure on a randomly chosen demand, but with such difficulty functions that a methodologically diverse two-out-of-three design would be worse than a one-methodology, diverse two-out-of-three design and even that the latter would be worse than a nondiverse two-out-of-three design.

It is also useful to point out that, when choosing diverse methodologies to build multiple versions, reducing correlation between failures of the versions is not the only consideration. The reliability to be expected of individual versions also matters. The theorems described in this section single out one of the factors determining the *pfd* of the redundant system, that is, the diversity in the difficulty functions of methodologies that are, on average, equally good (both examples and demonstrations are available in Littlewood and Miller [1989]).

To highlight the effects of other factors in practice, consider a manager who has to build a one-out-of-two system. One could choose a first methodology, A, which is suitable for the application and actually the best methodology in terms of the reliability it achieves on average. One might then look at other suitable methodologies, seeking one that is very diverse from A, in that its difficulty function is very different from that of A. Supposing that B satisfies this requirement, one would still have to consider the probability of joint failure for an AB system. Perhaps B tends to produce, for this application, programs that are on average much less reliable than those produced by A. It may then happen that B programs are most likely to fail on different demands from those on which A programs fail (i.e., A and B are truly very diverse), yet they are still likely to fail on these latter demands with a high enough probability that the *pfd* of an AB system is worse than that of an AA system. The dual situation is also possible: although A and B are the best methodologies, individually, for building this system, both an AA system and an AB system are worse than a CD system, built out of two methodologies, C and D, which are both inferior, on average, to both A and B but are more diverse. In conclusion, these theorems using simplified assumptions give qualitative indications on which project decisions are likely to improve the reliability of a diverse system, but actual predictions of such improvements would require a much more detailed analysis, and an amount of background statistical knowledge about the effects of the development method that is not usually available.

4.5. General Discussion on the Models

The EL model was a considerable advance in our understanding of probabilistic failure behavior of multiple diverse software versions. The idea of varying difficulty over the demand space provides an intuitively plausible mathematical rationale for failure dependence. The LM model somewhat softens the harshly pessimistic EL result. Nevertheless it leaves in place the single most important practical conclusion from this work, namely, that the level of dependence—particularly independence—between version failures cannot simply be assumed, but must be measured.

The approach used in these models (represented by the formalization of notions like variation of difficulty and forced diversity) seems quite powerful and generally applicable. For example, in Hughes [1987] and Littlewood [1996], the EL and LM models are generalized to account for common mode failures in nonsoftware-based systems. The approach casts doubt upon claims for failure independence even in functionally diverse systems [Littlewood et al. 1999]. In short, functional diversity is recommended by the need to reduce correlation among the physical failures of sensors, actuators, and transmission

chains, but this argument does not extend to a promise of independence in failures caused by design faults in the processing. An analysis similar to the ones in the previous section shows that the difficulty functions for the developers of the two versions must be considered. Functional diversity, as a way of tolerating design faults, must be seen as a special kind of forced design diversity, requiring positive evidence for any claim of low correlation between failures.

It should be emphasized that most of the results arising from the models concern averages over demand spaces, over populations of programs, and so on. Thus the difficulty function in EL, the probability that a randomly chosen program will fail on x, is an average over all programs. The reliabilities are averages over programs and demands. Actual realizations may differ from these averages. We have already noted that Knight and Leveson found their two-out-of-three systems an order of magnitude more reliable than the single versions on average, but there were particular single versions that were more reliable than particular two-out-of-three systems. In the absence of information about the particular (e.g., when taking an early design decision as to whether to use forced or unforced diversity) these average results give useful guidance. But they do not allow strong claims to be made for a particular system, for example, a particular system based upon forced diversity. In other words, they do not absolve us of the responsibility to evaluate what has actually been achieved.

In addition, the results all concern preferences—the theorems involve inequalities—and do not quantify the advantages of different approaches. Thus, the LM results that say that forced diversity is a good thing, do not tell us how much improvement will be delivered in a particular instance. Once again, this observation does not undermine the models' general recommendations for ways of best achieving reliability, but it provides further evidence that the problem of evaluating what has been achieved still remains.

5. PRACTICAL IMPLICATIONS OF RELIABILITY MODELS FOR DIVERSE SYSTEMS

5.1. Achievement of Reliability

The models described so far give a seemingly obvious indication for developers and managers: diversity is useful, and once diversity has been chosen, forcing diversity is better than just letting it happen willy-nilly. For instance, when choosing the methods to be used in a component activity of development (say, design specification, or testing) for two program versions, we should try to choose methods that create diverse difficulty functions for the people using them.[15] For instance, if we could identify two main classes of demands, D_1 and D_2, we would then try to choose two methods that, although equal in average quality, differ in the demand class on which they are most effective. If we first chose to build one version with method A, knowing that A is most effective on obtaining good reliability for demands of class D_1, we would then try to choose for the second version a development method that yields the same reliability as A on average, but is especially good for programming the response to demands of class D_2.

This simple example also shows the practical limitations of this qualitative advice.

—We seldom know the strengths and weaknesses of the different methods available. When we know them, it is usually in terms of the likelihood of people making mistakes, not of the likelihood that the defects caused by these mistakes cause failures in operation

[15] Diversity could even extend to the choice of staff, so that a certain role in one team is filled by a person whose main strength is in dealing with the demands of class D_A; the same role in the other team is given to a person with more expertise on demands of class D_B. The practical difficulties in this approach are obvious, although it may be feasible in some projects. On the other hand, practices such as having requirement reviews conducted by staff with diverse expertise are justified and, to an extent, can be guided, by considerations like these.

(which depends not only on the defects being present but also on the probabilities of the various demands).

—This advice is valid when all the methods from which we choose offer the same guarantees of reliability. When this is not true, we have to trade off the degree of diversity between versions against the risk of lower reliability in the version developed with the worse method. To know whether we would be better off with the two diverse methods or just using the better one for both versions, we would need to quantify their differences in some detail, and such quantitative knowledge is usually missing.

—Developing a system or program includes many interacting activities, so that the forms in which diversity could be applied are many. If we had, say, two programming languages L_1 and L_2 and two test methods T_1 and T_2, these two activities alone would allow four diverse combinations from which to choose. Knowledge of the pairwise diversities (between languages and between test methods) is not sufficient to know about the diversity among the four combinations.

—All these decisions are subject to practical constraints, which may help pruning the list of possible choices, but introduce additional criteria to be satisfied. For instance, the expertise of existing staff must be used; methods that have achieved particular results in other environments may not achieve the same in the environment of the current project; methods of similar quality may differ in cost, and thus differently affect the resources available for other activities.

In the end, all these problems indicate that we need a better ability to predict the effects of the means by which we try to achieve diversity. Developing better rules for using diversity depends on developing better models to predict and assess the effects of the specific ways of pursuing it.

5.2. Reliability of a Specific Two-Version System

5.2.1 Prediction for an Individual System: Distribution of pfd vs. Average pfd. The conceptual models presented in Section 4 describe what happens on average and help our understanding of the problems, but do not help us to evaluate a specific pair of versions.

In practice, when we deal with the specific pair of versions developed, we wish to know that the *pfd* of the pair[16] is lower than a certain bound, with a certain probability. In other words, since we know that the *pfd* of an individual pair may be very different from the average, we will need information about the probability distribution of the *pfd*, rather than just its average. The probability distribution provides the answers to questions of the form, "What is the probability that this particular pair has a satisfactorily low *pfd*?"

What can be said about design preferences in terms of distributions is rather simple. Given a single-version, one-out-of-two system, if we substitute one of the two channels with a different implementation, the *pfd* of the resulting system will obviously be no worse than that of the original, single-version system. If we consider, rather than an individual one-version and an individual two-version system obtained from it, the distributions of their *pfd*, it turns out that the two-version system would have both a better mean *pfd* and a better confidence level for any chosen upper bound.[17]

[16] We use the phrase "*pfd* of the pair," informally, to mean "probability of failure on demand of a one-out-of-two system built from this pair, with perfectly reliable adjudication."

[17] A way of stating these results is "diversity is *always* beneficial." This must be qualified: to build a one-out-of-two system (with perfect adjudication), choosing from a given population of possible versions, it is always better to choose two versions for the two channels, rather than the same version for both of them. If instead the choice is between a pair of versions built to a certain quality standard, and a single version built to a higher standard, the issue becomes one of cost-effectiveness and returns from additional effort spent on quality, as discussed in Section 3.

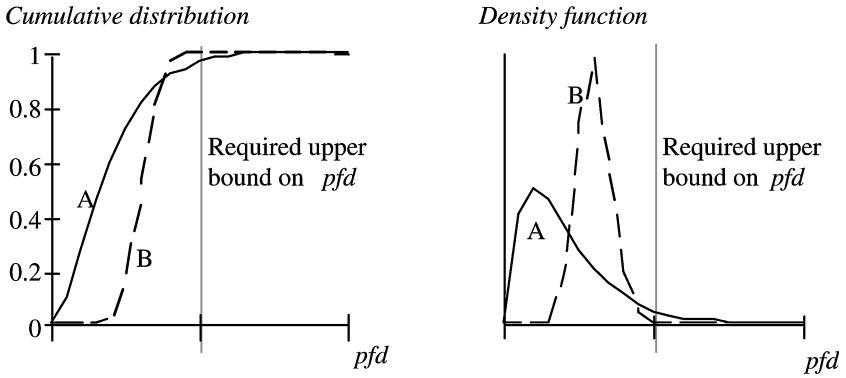

Fig. 5. Two hypothetical distributions of *pfd* for systems developed with different methods. Method B gives a higher (worse) expected value of *pfd*, but better confidence that the requirement on the *pfd* is satisfied.

It is appropriate to point out that a requirement of a high confidence level for a certain upper bound on the *pfd* may in principle require different design decisions from those required to achieve a low average *pfd*. For instance, the graph in Figure 5 shows two probability distributions A and B, such that A has the lower (better) average *pfd*, but B gives a higher confidence of a *pfd* smaller than a required bound. The "tail" of the density function for distribution B beyond the required bound has a smaller area than the tail of distribution A. In other words, system design decisions that produced distribution A would give a better average reliability over many produced systems; but system design decisions that produced distribution B would give a better probability of any individual system satisfying its requirements. No study so far has pointed to such counterintuitive situations, but their occurrence cannot be excluded.

More importantly, there are scenarios in which diversity produces only a small improvement in the average *pfd* of versions, but a more sizeable improvement in the requirements that can be satisfied with high confidence. For instance, the situation depicted in Table II would arguably be very satisfactory. In this scenario, the reliability requirement that can be satisfied with 99% probability is an order of magnitude better with a two-version system than with a one-version system. What kind of methodology for producing pairs of versions would be likely to produce such results? In terms of the population of the versions that can potentially be produced with the methodology, the numbers in Table II would indicate that the more reliable versions all tend to suffer from faults causing failures on the same set of demands (hence the small gain in the average *pfd* achieved by diversity); however, any additional faults present in the less reliable versions are spread over different demands, so that they are relatively unlikely to cause system failures in the two-version system.

These considerations show the importance of the whole distribution of the *pfd*. With the models introduced in Section 4, this distribution is completely determined by the difficulty functions, which characterize the methodologies for producing program versions, together with the demand profile (the probabilities of the various demands on the system) of the usage environment. However, the difficulty functions are unknown in practice. A recent line of research for improving this situation studies the distributions that can be expected in practice; early considerations on the effects of plausible distributions were reported in Popov et al. [1998]. Another aspect of great interest is how development processes may affect diversity, that is, what one may hope to know about these distributions, or notional

Table II. A Hypothetical Desirable Scenario for the Application of Diversity

	Mean *pfd*	99th Percentile (*pfd* That Can Be Certified with 99% Confidence)
Single-version system	10^{-3}	10^{-2}
Two-version system	$0.5\ 10^{-3}$	10^{-3}
Ratio improvement obtained by diversity	2	10

populations of versions from which the versions in a specific system has been extracted. This kind of knowledge, even in a qualitative or imprecise form, is of great relevance for design decisions. We return to this topic in Section 6.

In the next two sections, we study the reliability of a specific two-version system seen by itself, without any reference to notional populations of versions. We thus address the problem of assessing a two-version system, without any knowledge about the distributions that can be expected from the development methods used.

5.2.2 Completely Known Versions. As a first step, we can describe the *pfd* of a pair of versions A and B that are completely known: for each demand, we know whether it is a failure point for version A and/or for version B. This unrealistic case shows the way for developing more practical prediction procedures. We can describe this knowledge as a pair of *binary* functions $\omega_A(x)$ and $\omega_B(x)$ describing the behaviors of versions A and B, respectively, on the demand x. Saying that, for instance, $\omega_A(x) = 1$ means that version A deterministically fails on demand x. Then the probabilities of failure of versions A and B on a randomly selected demand X (probability of failure per execution) are:

$$P_A \equiv P(\text{A } \textit{fails on } X) = E(\omega_A(X)) = \sum_{x \in D} Q(x)\omega_A(x)$$

and

$$P_B \equiv P(\text{B } \textit{fails on } X) = E(\omega_B(X)) = \sum_{x \in D} Q(x)\omega_B(x),$$

where D denotes the demand space and $Q(x)$ is the probability that x will be input to the software (the demand profile of the software). For a specific demand x, the probability of common failure is then either 0 or 1:

$P(\text{A fails on } x \text{ and B fails on } x) = \omega_A(x)\ \omega_B(x)$

$$pfd_{AB} = P(\text{A and B fail on randomly chosen demand}) = \sum_{x \in D} Q(x)\omega_A(x)\omega_B(x). \quad (10)$$

It turns out that this expression can be written as

$$pfd_{AB} = P_A\ P_B + cov(\Omega_A, \Omega_B), \quad (11)$$

where the random variables Ω_A and Ω_B are defined as the values taken by ω_A and ω_B on a randomly chosen demand.

Equation (11) is very similar to (9) in Section 4.3, but (9) is based on the difficulty functions for two different development methodologies, which can take any value between 0 and 1 (representing the probability that a randomly chosen version, developed with that methodology, would fail on a given demand). Here, instead, we are describing two known versions. The functions $\omega_A(x)$ and $\omega_B(x)$ can only take the values 0 and 1, and the only uncertainty concerns the choice of the next demand x, described by the probability distribution $Q(x)$.

5.2.3 Using Results from Testing: Subdomains, Modes of Operation. The description given in the previous section would only be useful if one knew the behavior of each version on each possible demand, that is, for

each demand whether it is a failure point or a success point, for each version. This level of detailed knowledge is normally unattainable. The knowledge that can be obtained is at a much coarser level: by realistic testing, we can make predictions about the likelihood of each version failing on a randomly chosen demand. We can also specialize this knowledge slightly, by testing separately for separate classes of demands. Subdividing demands into classes is common practice for designers (e.g., in terms of *modes* of operation of a system) and software testers, who call these classes *subdomains* (in the demand space, often called the "input space" or "input domain" of a program). For instance, testers find it useful to define sub-domains on the basis of which function of the program (as defined in its requirements) the demands invoke, or on the basis of which parts of the code they cause to be executed.

Referring again to a system of two versions A and B, let us consider a division of the demand space into subdomains that completely cover the demand space, without any overlapping between them (a *partition* of the demand space). We call the subdomains themselves $S_1, S_2, \ldots, S_n$. We can define the probability of failure of a version when subjected only to demands from a specific subdomain, for example, $P(A \text{ fails} \mid S_i)$ will designate the probability that A fails on a demand chosen randomly from subdomain S_i, according to the probability distribution of demands in actual operation. We can then write the probability of common failure as

$$\begin{aligned} pfd_{\mathrm{AB}} &= P(\mathrm{A, B\ fail}) \\ &= \sum_i P(\mathrm{A, B\ fail} \mid S_i) P(S_i). \end{aligned} \tag{12}$$

The models described in Section 4 apply again within each subdomain; that is, in general,

$$\begin{aligned} &P(\mathrm{A, B\ fail} \mid S_i) \\ &\quad \neq P(\mathrm{A\ fails} \mid S_i) P(\mathrm{B\ fails} \mid S_i), \end{aligned} \tag{13}$$

Equality would only apply in special cases, for example, hardware-only versions that are subject only to physical failures and for which the stress to which they are subject is known to be constant across a certain class of demands. In most cases, one would expect a restricted class of demands to pose similar problems to the designers of two versions, so that the EL model would apply; in each subdomain, the left-hand term in (13) would be greater than the right-hand term. So, in practice, a regulator can use the sum in the left-hand side of the following expression,

$$\begin{aligned} &\sum_i P(\mathrm{A\ fail} \mid S_i) P(\mathrm{B\ fails} \mid S_i) P(S_i) \\ &\quad \leq \sum_i P(\mathrm{A, B\ fail} \mid S_i) P(S_i) = pfd_{\mathrm{AB}}, \end{aligned} \tag{14}$$

as a *claim limit* for the *pfd* of a two-version system.

Even if Formula (13) could be written with an equal sign (independent failures of the two versions, *conditional* on demands from a given subdomain) for all subdomains, this would not imply unconditional independence of failure. In terms of reliability estimates over subdomains, pfd_{AB} can be written, in a general form, as

$$\begin{aligned} pfd_{\mathrm{AB}} = {} & P(\mathrm{A\ fails}) P(\mathrm{B\ fails}) \\ & + cov1 + cov2, \end{aligned} \tag{15}$$

where the term $cov1$ is obtained by considering the *pfd* values of the two versions as functions of the subdomains, and taking their covariance over all the subdomains,

$$\begin{aligned} cov1 = {} & \sum_i (P(\mathrm{A\ fails} \mid S_i) - P(\mathrm{A\ fails})) \\ & \times (P(\mathrm{B\ fails} \mid S_i) - P(\mathrm{B\ fails})) P(S_i) \end{aligned} \tag{16}$$

and the term $cov2$ is obtained by computing the covariance of the Ω functions of the two versions in each subdomain, and

taking its average over all subdomains:

$$cov2 = \sum_i (cov(\Omega_1, \Omega_2 \mid S_i))P(S_i) = \sum_i \left(\sum_{x \in S_i} (\omega_A(x) - P(\text{A fails} \mid S_i))(\omega_B(x) - P(\text{B fails} \mid S_i))P(x \mid S_i) \right) P(S_i). \quad (17)$$

Each term in the inner sum above represents the difference between the two sides of inequality (13). Assuming each such term to be 0 (i.e., conditional independence within each subdomain) makes *cov*2 equal to zero.

There is a general similarity between the mathematics in Section 4 and in the parts of this section leading to Eqs. (11) and (15). In Section 4, the models described the expected behavior of two randomly chosen versions, given the difficulty functions that specify their likelihood of failing on individual demands. Equation (11) described failures of two specific versions, given detailed knowledge of whether they fail or succeed for every specific demand. Equation (15) again describes two specific versions, given coarser-grained knowledge about their failures for *classes* of demands, and again it can be shown that the probability of common failure is equal to the product of the versions' individual probabilities, plus covariance terms. All these cases are mathematically similar. Even given a knowledge that the two versions fail independently in every possible special condition (conditional independence on demands or classes thereof), the existence of variation between demands or classes thereof makes it impossible to deduce automatically that the versions fail independently for demands drawn from the whole demand space. Under the EL scenario of two versions drawn from the same distribution (i.e., developed with the same methodology, yielding similar difficulty functions over the demand space), we would expect the average pair of versions to show both positively correlated failures over each subdomain, and positively correlated probabilities of failure per subdomain, over the set of all subdomains. This latter property implies that the claim limit defined by inequality (14), once estimated in practice, is likely to be more stringent (to admit a weaker reliability claim, corresponding to a higher *pfd*) than the simpler limit given by $P(\text{A fails})\ P(\text{B fails})$.

Equation (15) describes essentially the same phenomenon as Eq. (11), only substituting the probability of failure over a subdomain for the deterministic failure on an individual demand. One can see that this expression applies for any possible subdivision of the history of demands on the system into disjoint subsets. Instead of subdividing the demands statically on the basis of the values of the sensor inputs, we could classify them on the basis of any other variable likely to affect the *pfd*, for example, modes of operation of the plant under which the demands originate.[18] A special case corresponds to the model described in Hughes [1987] and Littlewood [1996] for the failure of hardware-only systems, in which the subdomains are interpreted as conditions of operation producing different levels of stress on the redundant components, and the components themselves are assumed to fail independently, conditionally on the current operating condition.

5.3. Extension to Continuous-Operation Systems (Control Systems)

So far, we have described our versions in terms of their responses to discrete demands, which are chosen from the demand space in statistically independent ways. There are systems (typically continuous control systems) for which this description

[18] The difference is that, with this latter subdivision, two identical demands could be classified as belonging to different classes because they took place during segments of operation that are classified differently, for example, production vs. maintenance phases in a production plant, or even night shift vs. daytime shift. The same equations still hold, provided that the classification of the demands is consistent.

is difficult to apply, in that there is no natural subdivision of their execution histories into isolated, statistically independent demands. For these systems, the measure sought is often the reliability function in continuous time $R(t)$, rather than the *pfd*.

Reliability can be evaluated via testing, both for single-version systems and fault-tolerant systems considered as black boxes [Littlewood and Strigini 1998]. The basic problems are similar to those affecting assessment of on-demand systems discussed so far in this article. But when we try to study how the behaviors of diverse versions interact to produce the behavior of a diverse system, some new possibilities and difficulties arise. We summarize these here, referring the reader to Popov and Strigini [1998] for further information.

5.3.1. Pitfalls in Fine-Grain Modeling of Execution Sequences. For any (single- or multiple-version) design, there are two extreme ways of describing the execution of continuous control systems [Strigini 1996; Littlewood and Strigini 1998];

—as a special case of systems subject to independent demands, to which we can apply the models described in this article. As demands, we designate long periods of execution, either corresponding to whole missions (an aircraft's flight, the whole period of operation of a plant between two periods of inactivity), or simply long enough that the dependency between the software's behavior in two of these periods can be neglected; and

—in finer detail, by considering that each version repeatedly executes a "read sensors, process the readings, output results" cycle. So, a control system is seen as subject to a long series of nonindependent demands, where one demand is just one reading of the software's input variables. For this case, we say that each reading is "one input," and the successive inputs form a trajectory in an "input space," having as many dimensions as the number of input variables read by the software. Models assuming independence between successive steps have been published, and some have explored the effects of dependence between them [Bondavalli et al. 1999].

This second, more detailed approach runs into many problems in practical use. For instance, to be useful it requires one to describe the dependency between failures on successive inputs. There are many reasons for believing that these are positively correlated [Strigini 1996], but no simple way of estimating the degree of correlation. This problem is present even with a single-version system; when dealing with multiple versions, further problems appear. For instance, the models must represent the fact that each step of execution is affected not only by sensor inputs, but by the values of the software's internal variables. So, the values of internal variables must be considered as additional inputs that the versions read. But then the sequences of inputs read by two versions are not equal (not even approximately equal, since software faults may cause the internal variables of one version to take arbitrary incorrect values), and modeling how they are related becomes hopelessly difficult. In conclusion, for practical purposes of measurement and inference the first, coarser-grained of the two modeling options listed is the convenient one.

5.3.2. Medium-Grain Models: Transitions Between Modes of Operation with Different Reliability. We have seen that very detailed models of the versions' behavior over time (as opposed to responses to a single demand) are too difficult to apply, but we may hope to apply such models to coarser descriptions of the input sequences, as done in Section 5.2 with subdomains of the demand space. There are cases in which the reliability (or the failure rate) of the software can be estimated separately for different conditions of operation, corresponding to different regimes of operation of the controlled system or states of its environment. The evolution of the operating conditions can be modeled by a Markov chain or other stochastic models. Each state in the chain corresponds to a different operating

condition, characterized by a failure rate for (each version of) the software. Knowing these failure rates and the rate of common failures, for each condition of operation, it is then possible to predict the reliability of the software, and also to see how it would be affected by changes in the way conditions of operation alternate over time.

Once more, it can be shown that if these failure rates differ, any statement of independence conditional on one state in the Markov chain (i.e., one operating condition) does not extend to unconditional independence. Apart from this, no simple general rule can be stated. Both the covariance between the failure rates of the versions over the set of operating conditions and the rates of transition between these conditions affect the unconditional failure rate of the system. For instance, knowing that two versions exhibit independent failures on individual execution steps, conditional on the condition of operation, does not allow one to conclude anything about the correlation between their failures over a whole mission: the probability of both versions suffering a failure before the end of the mission may or may not be greater than the product of the probabilities for the two individual versions, depending on the detailed statistical properties of the succession of conditions of operation. Even though they give no simple general guidance, these models can be used for practical predictions on a specific system, if the various parameters are estimated by testing; *but in practice this estimation will usually be no less expensive than direct estimation of the system's reliability*. In conclusion, again, more detailed models than those described in this article have not produced, so far, appreciable benefits.

6. SUMMARY AND OPEN ISSUES

This survey has shown that assessing the reliability of diverse systems, and guiding decisions to engineer these systems, are hard problems. It is useful now to recall which solid knowledge is available, and which questions are open to research.

6.1. What Is Known

When planning to build a new system, we should expect a one-out-of-n system built with different versions to be substantially more reliable than one using multiple copies of one of the versions. This is a precise indication for builders of simple, parallel-redundant protection systems. Doubts arise in two forms:

—there are systems for which a diverse structure causes serious design difficulties compared to a nondiverse redundant structure. This complexity might offset the gain achievable from diversity. This problem will mostly be felt in majority-voted and/or active-control systems, rather than in simple one-out-of-N safety systems such as the one in Figure 4; and

—with a limited budget, it is uncertain whether concentrating all efforts on one version might produce higher reliability than dividing them over multiple diverse versions. This doubt is strongest in projects with great freedom effectively to tradeoff improvement efforts between versions. It is least strong in projects in which it is believed that no useful effort has been spared in making the individual versions as reliable as possible, and in those projects which are restricted to combining preexisting products.

When the time comes to assess the reliability achieved by a fault-tolerant system, the presence of diversity does not help much. In particular, we should not expect failures of diverse versions to be independent. Apart from experimental results to this effect, this statement is supported by conceptual models that are very widely applicable. These models predict that guaranteeing independence between the developments of two versions should be expected to yield positive correlation between failures of the versions. They also show a general direction for efforts towards reducing this correlation.

6.2. The Ubiquity of Variation of Difficulty

The EL and LM models described in Section 4 turn out to be applicable to describing many situations characterized by variation of some stress or unreliability factor over a space. Very similar or identical mathematical expressions turn out to be useful for modeling, for instance:

—the probability of common failure of two specific software versions, or more generally, any two diverse (hardware and/or software) channels (Section 5);

—the likelihood that design faults will remain undetected through two applications of (the same or two different) fault-finding [Littlewood et al. 2000]; and

—the likelihood of common mistakes by different persons performing the same task.

Any detailed understanding of how diversity is achieved through software or system development depends on modeling several such situations. In the development of a system, diversity of human errors among the teams performing the same stage of development activities on the diverse versions, diversity in the effect of human error-proneness under diverse development constraints, and diversity among the fault-removing activities applied to each version in sequence, all matter. Many of these detailed aspects of diversity have not been studied empirically yet, but it appears that any mathematical advance will be of immediate benefit in modeling all of them.

6.3. Open Questions and Research in Progress

In practice, design decisions and reliability assessment are interdependent activities. A requirement on designs is that the system they produce should be easy to assess, and reliability assessment relies in part on what is known about the performance of the design and development methods employed. Research is needed on both these aspects:

—for reliability assessment: improving on current methods for assessment, taking advantage as fully as possible of all system-specific information both about the design decisions and about the results of product-based validation;

—for reliability achievement: suggesting directives or decision criteria for design and project management to improve the reliability of diverse systems, for example, advice on how best to "force" diversity, issues of trade-offs between individual version reliability and diversity between versions.

The two goals are interrelated, in that any rational justification of design and management decisions must be based on some kind of ability to forecast their effects. In more detail:

—the task of assessment has two sides:
- predicting the effects of decisions made in development on achieved reliability. Research here has to explore the empirical evidence available about these effects in the generality of systems and build predictive models linking observable characteristics of a specific development process and its product to future reliability. These models are useful for achieving reliability, by indicating which decisions are likely to improve results, even when their predictions are too imprecise for assessment of the finished product. For reliability assessment, their role is, formally speaking, to produce prior probabilities for Bayesian inference from direct, system-specific evidence of reliability and informally, to give approximate but useful indicators of the reliability range to be expected; and
- inferring future reliability from direct evidence (testing and operation). Here, Bayesian inference provides a sound framework, but research is needed to make it useful in practice. Two requirements are to ensure that the necessary calculations are feasible, possibly by finding suitable approximation methods, and to produce guidance in choosing priors, via models as described above and via

methods for obtaining satisfactory approximations and bounds on the required predictions;

—findings about methods for reliability assessment also affect reliability achievement by suggesting which decisions will make it easier to assess the reliability of the resulting product against its requirements. This applies even to assessment methods that are strictly a posteriori, and thus cannot inform about which decision will produce better reliability.

All this research depends on three sources of knowledge: experimental evidence in its two forms of experience from real-world projects and from controlled experiments, and theoretical modeling. Experimental evidence is essential, but both its sources are severely limited because of costs. Real-world systems are few and heterogeneous, and fewer data are available about them than would be desirable. Controlled experiments cannot achieve both realism and statistical significance at affordable cost. They must therefore be focused to address specific conjectures, which if validated can be combined via theoretical modeling to produce results of practical relevance. In short, no practical directive for designers and assessors can be expected from experimental evidence alone. Theoretical work is necessary to distill sparse statistical data into conjectures, direct the testing of these conjectures, and combine the resulting knowledge into useful practical directives. Among other benefits from theoretical work, there is a hope of gleaning useful information for diversity research from the data that have been collected in experiments aimed at other problems in software engineering or cognitive psychology.

REFERENCES

ADAMS, E. N. 1984. Optimizing preventive service of software products. *IBM J. Res. Devel. 28*, 1, 2–14.

AMMANN, P. E. AND KNIGHT, J. C. 1988. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput. C-37*, 4, 418–425.

ANDERSON, T., BARRETT, P. A., HALLIWELL, D. N. AND MOULDING, M. R. 1985. An evaluation of software fault tolerance in a practical system. In *Proceedings of the 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*. (Ann Arbor, MI.)

BABBAGE, C. 1974. On the mathematical powers of the calculating engine (unpublished manuscript, December 1837). In *The Origins of Digital Computers: Selected Papers*, B. Randell, Ed. Springer-Verlag, New York, 17–52.

BISHOP, P. G. 1988. The PODS diversity experiment. In *Software Diversity in Computerized Control Systems*, U. Voges, Ed. Springer-Verlag, New York, pp. 51–84.

BISHOP, P. G. AND PULLEN, F. D. 1988. PODS revisited—A study of software failure behavior. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*. (Tokyo), IEEE Computer Society Press, Los Alamitos, Calif.

BLOUGH, D. M. AND SULLIVAN, G. 1990. A comparison of voting strategies for fault-tolerant distributed systems. In *Ninth Symposium on Reliable Distributed Systems* (*SRDS-9*) (Huntsville, AL), IEEE Computer Society.

BONDAVALLI, A., CHIARADONNA, S., DI GIANDOMENICO, F. AND STRIGINI, L. 1999. A contribution to the evaluation of the reliability of iterative-execution software. *Soft. Test. Verif. Reliab. 9*, 3, 145–166.

BRIERE, D. AND TRAVERSE, P. 1993. Airbus A320/A330/A340 electrical flight controls—A family of fault-tolerant systems. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing* (*FTCS-23*). (Toulouse, France), IEEE Computer Society, Los Alamitos, Calif.

DI GIANDOMENICO, F. AND STRIGINI, L. 1990. Adjudicators for diverse-redundant components. In *Ninth Symposium on Reliable Distributed Systems* (*SRDS-9*) (Huntsville, AL.), IEEE Computer Society Press, Los Alamitos, Calif.

DYER, M. 1992. The Cleanroom Approach to Quality Software Development. *Software Engineering Practice*. Wiley, New York.

ECKHARDT, D. E. AND LEE, L. D. 1985. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. Softw. Eng. SE-11*, 12, 1511–1517.

FAA 1985. Federal Aviation Administration, Advisors Circular AC 25 1309–1A.

HAGELIN, G. 1988. ERICSSON safety systems for railway control. In *Software Diversity in Computerized Control Systems*, U. Voges, Ed. Springer-Verlag, New York, pp. 11–21.

HUANG, Y., KINTALA, C., KOLETTIS, N. AND FULTON, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *25th International Symposium on Fault Tolerant Computing* (*FTCS-25*) (Pasadena), IEEE Computer Society Press, Los Alamitos, Calif.

HUGHES, R. P. 1987. A new approach to common cause failure. *Reliab. Eng. 17*, 211–236.

KANTZ, H. AND KOZA, C. 1995. The ELEKTRA railway signalling-system: Field experience with an actively replicated system with diversity. In *Proceedings of the 25th IEEE Annual International Symposium on Fault -Tolerant Computing (FTCS-25)* (Pasadena), IEEE Computer Society Press, Los Alamitos, Calif.

KERSKEN, M. AND SAGLIETTI, F. Eds. 1992. Software fault tolerance: Achievement and assessment strategies. Research Reports ESPRIT, Springer-Verlag, New York.

KNIGHT, J. C. AND AMMAN, P. E. 1985. An experimental evaluation of simple methods for seeding program errors. In *Proceedings of the Eighth International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, Calif.

KNIGHT, J. C. AND LEVESON, N. G. 1986. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Softw. Eng. SE-12*, 1, 96–109.

KNIGHT, J. C. AND LEVESON, N.G. 1990. A reply to the criticism of the Knight & Leveson experiment. *ACM SIGSOFT Softw. Eng. Notes 15*, 1, 24–35.

LAPRIE, J. C., ARLAT, J., BEOUNES, C. AND KANOUN, K. 1990. Definition and analysis of hardware-and-software fault-tolerant architectures. *IEEE Comput. 23*, 7, 39–51.

LARYD, A. 1994. Operating experience of software in programmable equipment used in ABB Atom nuclear I&C application. In *Advanced Control and Instrumentation Systems in Nuclear Power Plants. Design, Verification and Validation. IAEA/IWG/ATWR & NPPCI Technical Committee Meeting* (Espoo, Finland).

LEE, I. AND IYER, R. K. 1995. Software dependability in the Tandem GUARDIAN system. *IEEE Trans. Softw. Eng. 21*, 5, 455–467.

LINDEBERG, J. F. 1993. The Swedish state railways' experience with *n*-version programmed systems. In *Directions in Safety-Critical Systems*, F. Redmill and T. Anderson, Eds. Springer-Verlag, New York, p. 36.

LITTLEWOOD, B. 1996. The impact of diversity upon common mode failures. *Reliab. Eng. Syst. Safety*. *51*, 101–113.

LITTLEWOOD, B. POPOV, P., STRIGINI, L. AND SHRYANE, N. 2000. Modelling the effects of combining diverse software fault removal techniques. *IEEE Trans. Softw. Eng. SE-26*, 12, 1157–1167.

LITTLEWOOD, B. AND MILLER, D. R. 1989. Conceptual modelling of coincident failures in multi-version software. *IEEE Trans. Softw. Eng. SE-15*, 12, 1596–1614.

LITTLEWOOD, B. AND STRIGINI, L. 1993. Validation of ultra-high dependability for software-based systems. *Communi. ACM*, *36*, 11, (Nov.), 69–80.

LITTLEWOOD, B. AND STRIGINI, L. 1998. Guidelines for the statistical testing of software. Centre for Software Reliability, City University, London.

LITTLEWOOD, B., POPOV, P., AND STRIGINI, L. 1999. A note on reliability estimation of functionally diverse systems. *Reliab. Eng. Syst. Safety*. *66*, 93–95.

LYU, M. R. Ed. 1995. *Software Fault Tolerance*. Wiley, New York, 337.

LYU, M. R. Ed. 1996. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, New York.

MIGNEAULT, G. E. 1982. The Cost of Software Fault Tolerance Technical Report. NASA Langley Research Center, Hampton, Va.

MoD, 1996. Safety management requirements for defence systems. U.K. Ministry of Defence.

MoD, 1997. Requirements for safety related software in defence equipment. U.K. Ministry of Defence.

MONGARDI, G. 1993. Dependable computing for railway control systems. In *Third IFIP International Working Conference on Dependable Computing for Critical Applications* (*DCCA-3*) (Mondello, Italy).

MUSA, J. D. 1993. Operational profiles in software-reliability engineering. *IEEE Softw.* (March), 14–32.

NICOLA, V. F. AND GOYAL, A. 1990. Modeling of correlated failures and community error recovery in multiversion software. *IEEE Trans. Softw. Eng. 16*, 3, 350–359.

POPOV, P. T. AND STRIGINI, L. 1998. Conceptual models for the reliability of diverse systems—New results. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing* (*FTCS-28*) (Munich, Germany) IEEE Computer Society Press, Los Alamitos, Calif.

POPOV, P., STRIGINI, L., AND PIZZA, M. 1998. The efficacy of diverse redundancy against design error: Some practical considerations. In *Preprints of the INucE Third International Conference on Control and Instrumentation in Nuclear Installations* (Edinburgh).

RTCA/EuroCAE, 1992. DO-178B, Software considerations in airborne systems and equipment certification.

SHOOMAN, M. 1996. Avionics software problem occurrence rates. In *ISSRE'96, Seventh International Symposium on Software Reliability Engineering* (White Plains, NY).

SMITH, I. C., WALL, D. N., AND BALDWIN, J. A. 1991. DARTS—An experiment into cost of and diversity in safety critical computer systems. In *IFAC/IFIP/EWICS/SRE Symposium on Safety of Computer Control Systems (SAFECOMP '91).* (Trondheim, Norway), Pergamon Press.

STRIGINI, L. 1996. On testing process control software for reliability assessment: The effects of correlation between successive failures. *Softw. Test. Verif. Reliab. 6*, 1, 36–48.

TRAVERSE, P. J. 1988. AIRBUS and ATR system architecture and specification. In *Software Diversity in Computerized Control Systems*, U. Voges, Ed. Springer-Verlag, New York, pp. 95–104.

TURNER, D. B., BURNS, R. D., AND HECHT, H. 1987. Designing micro-based systems for fail-safe travel. *IEEE Spectrum 24*, 2, 58–63.

VOGES, U. AND GMEINER, L. 1979. Software diversity in reactor protection systems: An experiment. In *IFAC Workshop, SAFECOMP'79* (Stuttgart, Germany May 16–18).

VOGES, U. Ed. 1988. *Software Diversity in Computerized Control Systems.* Dependable Computing and Fault-Tolerance Series. Springer-Verlag, Wien, Austria.

VOGES, U. 1994. Software diversity. *Reliab. Eng. Syst. Safety 43*, 2, 103–110.

YEH, Y. C. B. 1998. Design considerations in Boeing 777 fly-by-wire computers. In *Third IEEE High-Assurance Systems Engineering Symposium* (*HASE*). (Washington, DC) IEEE Computer Society Press, Los Alamitos, Calif.

Received February 1999; revised February 2000; accepted May 2000