Popov, P. T. & Littlewood, B. (2004). The effect of testing on reliability of fault-tolerant software. Paper presented at the 2004 International Conference on dependable systems and networks, 28 June - 1 July 2004, Florence, Italy.

CITY UNIVERSITY LONDON

City Research Online

The Effect of Testing on Reliability of Fault-Tolerant Software

Peter Popov, Bev Littlewood
Centre for Software Reliability, City University
Northampton Square, London, EC1V 0HB
Phone: +44 (0) 207 040 8963
Fax: +44 (0) 207 040 8585
Email: ptp,b.littlewood@csr.city.ac.uk

Abstract

Previous models have investigated the impact upon diversity - and hence upon the reliability of fault-tolerant software built from 'diverse' versions - of the variation in 'difficulty' of demands over the demand space. These models are essentially static, taking a single snapshot view of the system. In this paper we consider a generalisation in which the individual versions are allowed to evolve - and their reliability to grow - through debugging. In particular, we examine the trade-off that occurs in testing between, on the one hand, the increasing reliability of individual versions, and on the other hand the possible diminution of diversity.

1. Introduction

The first major breakthrough in stochastic modelling of diversity came in a paper by Eckhardt and Lee [1]. The key idea here was that the different demands that a program might receive during operation would vary in 'difficulty' - specifically the probability of failure upon execution of a demand would be different for different demands[1]. From this insight it is possible to show that if more than one version executes each demand, the failures of the versions *cannot* be independent: in fact they will exhibit dependence in failure behaviour which ensures that fault tolerant systems built from the versions will be less reliable, on average, than if independence could be assumed.

The more variation in difficulty across demands, the worse becomes the problem: everything depends upon a key variance term. However, it is difficult to estimate the necessary parameters to answer questions such as 'how much worse is this system than it would be under independence?' The model is thus essentially a conceptual one. Nevertheless, the result is extremely important, with serious implications for industrial practice, since it precludes the use in software diversity of the simplistic probability models that have had some success in hardware reliability.

In the original version of the model (EL model), the intention was to represent the case where diversity occurred naturally as a result of the software development teams being given a free hand to create their versions 'independently' of one another. The model was later generalised [2] to take account of *forced diversity* (LM model), where the different teams would be required by a higher design authority to use different methodologies: for example different languages, testing and analysis techniques, etc. Here the intent was to exploit the possibility that what may be 'difficult' for one methodology (i.e. have a high propensity of failure) may be easy for another. In this model, it can be shown that it is no longer inevitable that a fault tolerant system will be less reliable than would be the case if the versions failed independently. Everything here depends upon a covariance factor, representing the dependence between the versions created from the different methodologies. In the event that this is negative, the reliability of a system built from the versions will be *better* even than it would be if the versions could be assumed to fail independently. Even when, as seems likely, the covariance is not negative, it can be shown that the Eckhardt and Lee result is the worst case that can occur.

Both these models are essentially static: they address questions concerning the reliabilities of unchanging versions, and the dependencies of failure behaviour of such versions. In the present paper we present a more general model that treats the case in which the versions are changing as a result of faults being found (and removed) in testing or in operational use. This extension is important because issues of cost effectiveness are at the heart of those parts of software engineering that are concerned with achieving system dependability. The work here is intended to deepen the understanding of the trade-offs that take place in building design-diverse fault tolerant systems. In particular, we investigate the impact

[1] In the original papers [1] and [2] the term 'input' was used instead of 'demand'. Here we refer to demand to avoid the confusion surrounding the term 'input' used in different contexts. A demand can consist of a single or many inputs to the software depending on the system context.

upon system reliability of improvements in version reliability, as these improvements also impact upon the degree of version dependence.

We begin by recalling the notation and models introduced in [1] and [2], which we are to extend.

For a particular set of requirements there is a population of all possible programs (versions) which (conceptually, at least) could be written, $\wp = \{\pi_1, \pi_2, \pi_3, ...\}$.

Many, if not most, of these programs will be incorrect, i.e. they sometimes give wrong output. An actual product development is then the random selection of π from $\wp$, i.e., the program is a random variable Π, with $P(\Pi = \pi) = S(\pi)$, for some measure $S(\bullet)$ over $\wp$. The measure $S(\bullet)$ can be thought of as representing the development methodology used.

Execution of a program version involves random selection of a demand from the demand space $F = \{x_1, x_2, ...\}$. That is, the demand is a random variable X with $P(X=x) = Q(x)$ for some measure $Q(\bullet)$ over F. Here $Q(\bullet)$ could be thought of as the usage distribution over demands. It might vary from one user environment to another.

The failure behaviour of the program is described by the score function

$$\upsilon(\pi, x) = \begin{cases} 1, \text{ if program } \pi \text{ fails on } x; \\ 0, \text{ if program } \pi \text{ does not fail on } x. \end{cases}$$

Thus the random variable $\upsilon(\Pi, X)$ represents the performance of a random program on a random demand: this is a model for the uncertainty both in software *development* and *usage*.

A key average performance measure is

$$\theta(x) = \sum_{\wp} \upsilon(\pi, x).S(\pi) = \mathrm{E}_S\left(\upsilon(\Pi, x)\right) \quad (1)$$

which is the probability that a randomly chosen program fails for a particular demand x. The heart of Eckhardt and Lee's idea is the recognition that $\theta(x)$ will generally take different values for different x, representing the varying 'difficulty' in correctly processing different demands. For a randomly chosen demand X, $\theta(X)$ is a random variable. Finally,

$$\mathrm{E}(\theta(X)) = \\ = \sum_{\wp}\sum_{F} \upsilon(\pi, x).S(\pi).Q(x) = \mathrm{E}_{S,Q}\left(\upsilon(\Pi, X)\right) \quad (2)$$

represents the probability that a randomly chosen program fails on a randomly chosen demand. In the presence of uncertainties of both development and usage, this represents the likelihood that our software fails.

Suppose now that we create, independently, two program versions. That is, we select Π_1 and Π_2 independently from $\wp$. These are *truly independent* in the conventional statistical sense:

$$P(\Pi_1 = \pi_1, \Pi_2 = \pi_2) = P(\Pi_1 = \pi_1).P(\Pi_2 = \pi_2). \quad (3)$$

It then follows that the probability that both Π_1 and Π_2 fail on a given demand x is:

$$P(\Pi_1 \text{ fails on } x, \Pi_2 \text{ fails on } x) = \\ \sum_{\wp}\sum_{\wp} \upsilon(\pi_1, x).\upsilon(\pi_2, x).P(\Pi_1 = \pi_1).P(\Pi_2 = \pi_2) \quad (4)$$

$$= P(\Pi_1 \text{ fails on } x).P(\Pi_2 \text{ fails on } x) = \left(\theta(x)\right)^2.$$

The conditional form of the joint behaviour is:

$$P(\Pi_2 \text{ fails on } x \,|\, \Pi_1 \text{ failed on } x) = \\ \frac{P(\Pi_1 \text{ failed on } x, \Pi_2 \text{ fails on } x)}{P(\Pi_1 \text{ failed on } x)} = \quad (5) \\ P(\Pi_2 \text{ fails on } x) = \theta(x).$$

Thus we can see that independently developed programs fail independently when executing a given fixed demand x.

However the situation is different when there is uncertainty concerning the demand, i.e., the programs execute a *random* demand, X:

$$P(\Pi_1 \text{ and } \Pi_2 \text{ both fail on } X) \\ P(\upsilon(\Pi_1, X)\upsilon(\Pi_2, X) = 1) = \\ \sum_{F}\sum_{\wp}\sum_{\wp} \upsilon(\pi_1, x)\upsilon(\pi_2, x)S(\pi_1)S(\pi_2)Q(x) = \quad (6)$$

$$\sum_{F} \left(\theta(x)\right)^2 Q(x) = \mathrm{E}\left(\Theta^2\right) = Var(\Theta) + \left(\mathrm{E}(\Theta)\right)^2 =$$

$$Var(\Theta) + \left(P(\Pi \text{ fails on } X)\right)^2.$$

(here the random variable $\Theta = \theta(X)$). The conditional form of the joint behaviour is:

$$P(\Pi_2 \text{ fails on } X \,|\, \Pi_1 \text{ failed on } X) = \\ \frac{Var(\Theta)}{E(\Theta)} + P(\Pi_2 \text{ fails on } X) \geq P(\Pi_2 \text{ fails on } X). \quad (7)$$

Equality holds if and only if $\theta(x) = \theta$ identically for all x and it seems likely that this will never be the case. This is the main result of [1]: that the failure behaviour of diverse versions will necessarily be *worse* than what could be expected under the assumption of independence, *even though the versions themselves truly are 'independent objects'* in the sense of (3).

The work reported in [2] extends this model to the case where several development methodologies A,B,C, etc. are available. These might represent, for example, different development environment, different types of programmers, different languages, different testing regimes, etc.

Each methodology induces a measure on $\wp$, the set of all possible program versions. A random program Π_A developed using methodology A will be version π with probability:

$$P(\Pi_A = \pi) = S_A(\pi).$$

If the methodologies are very diverse, we would expect a program with a high probability of selection under one methodology to have a low, perhaps zero, probability of selection under others. When methodologies are identical, they have identical measures $S(\bullet)$.

Within a particular methodology, the situation is exactly like that in [1]. Thus $\theta_A(x)$ is the probability of a randomly chosen program from methodology A failing on demand x; the random variable $\Theta_A = \theta_A(X)$ is the probability of Π_A failing on the random demand X, etc.

Consider two random program versions Π_A and Π_B developed independently under methodologies A and B respectively, i.e.,

$$P(\Pi_A = \pi_A, \Pi_B = \pi_B) = P(\Pi_A = \pi_A)P(\Pi_B = \pi_B) = S_A(\pi_A)S_B(\pi_B). \quad (8)$$

It follows that, for any given demand, x, two randomly chosen programs fail independently, as in (5). However, the probability of simultaneous failure on a *randomly chosen* demand X is:

$$P(\Pi_A \ fails on \ X, \Pi_B \ fails on \ X) = \mathrm{E}(\Theta_A\Theta_B) = Cov(\Theta_A, \Theta_B) + \mathrm{E}(\Theta_A)E(\Theta_B) = Cov(\Theta_A, \Theta_B) + P(\Pi_A \ fails on \ X)P(\Pi_B \ fails on \ X). \quad (9)$$

The conditional form of joint behaviour is:

$$P(\Pi_B \ fails on \ X \mid \Pi_A \ failed on \ X) = \frac{\mathrm{E}(\Theta_A\Theta_B)}{\mathrm{E}(\Theta_A)} = \frac{Cov(\Theta_A, \Theta_B)}{\mathrm{E}(\Theta_A)} + \mathrm{E}(\Theta_B) \quad (10)$$

This is greater than $\mathrm{E}(\Theta_B) = P(\Pi_B \ fails \ on \ X)$ if and only if $\mathrm{Cov}(\Theta_A, \Theta_B) > 0$. But since it is possible that $Cov(\Theta_A, \Theta_B) < 0$, it follows that using different design methodologies it is possible in this model to do even better than the (unattainable) goal of independent performance of versions in the single methodology case. This is the main result in [2].

2. Modelling the testing process

All this earlier work considers only a snapshot at a single moment in time: it concerns the joint failure behaviour of versions upon a single demand. In the present paper we shall extend these ideas to the case where the programs are allowed to increase in reliability as a result of the faults being identified and removed via testing. Clearly, when we come to build fault-tolerant systems from diverse versions, the overall system reliability will depend upon *both* the level of diversity achieved and the individual reliability of the versions. It is intuitively obvious that certain types of testing - for example back-to-back testing - will allow version reliability to improve only at the price of decreasing diversity.

Demands used for testing software are drawn from the *demand space* which is typically very large. Within this space a set of points (failure regions) will be associated with a fault: typically there will be many demands that would trigger a particular fault (or, conversely, removing a fault will result in many demands, which previously could not be executed correctly, being transformed into ones that can). A decision mechanism judges the executions of demands by software as acceptable or failed. Finally the programmer should respond to detected failures by trying to identify the faults causing the failures and remove them from software. The testing thus includes: i) a sequence of demands on which software is executed (a test suite), ii) a judging mechanism (for example oracle(s)) and iii) actions aimed at removing fault(s) causing a failure to occur. Clearly, the judging mechanism can itself be fallible, and so can the actions taken to remove a fault that is found. The efficacy of the testing process thus will depend upon both of these, as well as upon the fault-revealing power of the test suite.

Test suites are drawn in accord with the testing goal. If operational reliability is targeted the test suites are generated using the expected operational profile (probability distribution on demand space) of software. If debugging is targeted the test suite is generated according to what the debugger believes maximises the chances of finding faults. Usually, the size of the test suite (the number of demands included in it) is determined with respect to some *stopping rule* which gives the tester sufficiently high confidence that the goal (e.g. targeted reliability) has been achieved [3].

Clearly, with a given selection criterion a multitude of test suites can be generated, each being a *particular realisation* of a given test suite generation procedure. The same test suite may, in principle, be generated with different generation procedures. As a rule, the likelihood that a particular test suite will be generated will vary with generation procedures: this uncertainty will be modelled here using probabilities.

With regard to the judging mechanism even if the same test suite is used the effect of the testing on the same software version may vary if the detection and the fault removal are not perfect. Indeed if the detection mechanisms are different it may be that executing the same test suite with a particular software once may lead to many failures being detected, while in some other cases to fewer (in the extreme case none). The faults causing the undetected failures will not be removed. Similarly imperfect fault fixing may only partially remove the causing fault and in the worst case even introduce new faults. In summary, imperfection of failure detection and fault fixing will create uncertainty about faults remaining in the tested software and hence about its reliability. For these reasons we shall concentrate on the case where there is perfect detection and perfect fixing.

3. Testing with a perfect oracle and perfect fault-fixing

In this section we assume that both the failure detection and the fault-removal are perfect in which case the effectiveness of the testing is limited by the revealing power of the test suite used.

Let us define the set of all test suites, $\Xi=\{t_1,t_2,...\}$, which can be generated with a given generation procedure together with the probabilistic measure, $M(\bullet)$, defined on Ξ. $M(t)$ = P(T=t) is the probability that a particular test suite t is created by a random application of the test generation procedure. We use T to denote a test suite randomly chosen from Ξ using $M(\bullet)$.

We can model the effect of the testing using the scores of the tested software. We define the score on demand x of a *particular software version*, π, tested with a particular test suite, t, selected from Ξ, similarly to how the scores were defined by Eckhardt and Lee [1]:

$$\upsilon(\pi,x,t)=\begin{cases}1, \text{ if } \pi \text{ tested on } t \text{ fails on } x,\\ 0, \text{ if } \pi \text{ tested on } t \text{ does not fail on } x.\end{cases} \quad (11)$$

We use the notation $\upsilon(\pi,x,\varnothing)$ for the score of π on x *before* testing, i.e. $\upsilon(\pi,x,\varnothing)$ is identical to Eckhardt and Lee notation, $\upsilon(\pi,x)$. $\upsilon(\Pi,X,T)$ is the notation used for the score on a randomly selected demand, X, of a randomly selected version, Π, after being tested with a randomly chosen test suite, T.

Note that, since we have assumed perfect both the failure detection and the fault fixing, no new faults can be introduced during testing. Consequently, i) a value '0' of the score $\upsilon(\pi,x,\varnothing)$ implies '0' of the score $\upsilon(\pi,x,t)$, too; ii) '1' of the score $\upsilon(\pi,x,t)$ implies that the score before testing, $\upsilon(\pi,x,\varnothing)$, was also 1. The only possible difference between the score before and after testing is $\upsilon(\pi,x,\varnothing)$ = 1, $\upsilon(\pi,x,t)$ = 0. In other words, $\upsilon(\pi,x,\varnothing) \geq \upsilon(\pi,x,t)$. Clearly, it is sufficient for such a change that x belong to the test suite. Indeed, under the assumed perfection of detection and fault removal once x is executed by software the failure will be detected and the corresponding fault fixed with certainty.

Note that the assumed perfection of fault fixing implies fixing *all* faults that cause a failure on x. The inclusion of x in the test suite, however, is not necessary for the score on x to change from 1 to 0. Assume that a failure on x in the untested version is caused by a set of faults $O_x = \{f_1, f_2, \ldots\}$. Assume further that the faults in O_x cause failures on a set of demands $DX = \{x_1, x_2, \ldots\}$ different from x. Finally, let us assume that removing the faults from O_x will transform the demands from DX from failures to successfully processed demands. Clearly, if x is in the test suite all demands from DX will be successfully executed in the tested version. Similarly, if at least one demand y from DX is in the test suite it will cause a failure detection and then all faults from O_x will be fixed. As a result the demand x will be processed correctly in the tested versions, i.e. $\upsilon(\pi,x,t)$ = 0. More refined arrangements are also possible for $\upsilon(\pi,x,t)$ = 0 which do not require either of the demands from DX to be in the test suite. If all faults from O_x are fixed during testing because different sub-sets of O_x are found to cause failures on demands which are not in DX the tested software will execute successfully x and all other demands from DX. In summary, the tested software will have more demands converted from failures to successes than the number of failures observed during the testing.

Note that the score of the tested version is a function of the applied testing suite. If multiple versions are tested on the same test suite, clearly, the scores of all tested versions will be 0 on demands included in the test suite but may differ on demands not included in the test suite. Now we express the probability of failure of a particular version π on x with a randomly selected test suite, T, which is represented by the expected value of the score function $\upsilon(\pi,x,T)$ over the population of the tests:

$$\varsigma(\pi,x)=\sum_{\Xi}\upsilon(\pi,x,t).M(t) \quad (12)$$

Similarly, we can take the average over the population of programs, $\wp$:

$$\xi(x,t)=\sum_{\wp}\upsilon(\pi,x,t).S(\pi), \quad (13)$$

is the probability that a randomly chosen program, Π, tested with a particular test t fails on x.

This last probability represents how efficient a particular test is with respect to x when applied to the population of tested programs – the lower the probability the better. If the test leads to the removal of all faults from all versions, $\xi(x,t)=0$. $\xi(x,T)$, as a rule, will vary between test suites. Clearly $\xi(x,T)$ is related to the difficulty function of the *untested* population, $\theta(x)$. Since $\upsilon(\pi,x,\varnothing) \geq \upsilon(\pi,x,t)$ then $\theta(x) \geq \xi(x,T)$, too.

Finally:

$$\eta(\pi,t)=\sum_{F}\upsilon(\pi,x,t).Q(x)$$

is the probability that program π, tested on t fails on a randomly selected demand X.

Now, the mixed moment of the score over the population of programs and of test suites can be expressed as:

$$\zeta(x)=\sum_{\wp}\varsigma(\pi,x).S(\pi)=\sum_{\Xi}\xi(x,t).M(t)=\mathrm{E}_{\Xi,\wp}[\upsilon(\Pi,x,\mathrm{T})] \quad (14)$$

and represents the probability that a *randomly selected program*, Π, tested on a *randomly selected test suite*, T,

fails on a particular demand *x*. $\zeta(x)$ is the counterpart *after testing* of $\theta(x)$, the Eckhardt and Lee 'difficulty' function for the *untested* versions given in (1) above.

It is interesting to compare $\theta(x)$ and $\zeta(x)$. Since $\theta(x) \geq \xi(x,T)$, i.e. the difficulties are ordered on every demand, then $\theta(x) \geq \zeta(x)$ whatever $M(\bullet)$. The efficiency of the test generation procedure can be measured by comparing by how much $\theta(x) \geq \zeta(x)$. Intuitively, the bigger the difference the more efficient the testing (but this may be misleading since if $\theta(x) = 0$ there is no room for improvement).

If one had two testing procedures with $M_1(\bullet)$ and $M_2(\bullet)$ and their corresponding $\zeta_1(x)$ and $\zeta_2(x)$ one would be interested to know which of the two procedures is more efficient. If $\zeta_1(x) > \zeta_2(x)$ then the second procedure is more efficient than the first one and vice versa.

Yet another important question is whether variability of the difficulty changes as a result of the testing. In the extreme case it may be possible through testing to *compensate completely* for the variability of difficulty function, i.e. make $\zeta(x) = const$ across demands. Then the problematic assessment of the probability of joint failure of two version system (7) will be avoided since in this case unconditional independence of version failures will follow. At the very least it seems desirable to reduce the variability of $\zeta(x)$. Even though such a possibility exists it is unclear how realistic it is to expect it in practice. The constraints which must be imposed upon the testing so that the variability of the difficulty decreases are not known and open for future research.

The other extreme case, increase of variability as a result of the testing, is also possible.

With *forced diversity* [2] the likelihood to draw the same version from *different methodologies*, A and B, will be different. The expectations, (12) – (14), given above, can be similarly defined for each of the methodologies. For instance, the probabilities of failure on demand *x* of a program tested with *t*, taken at random from A and B respectively are $\xi_A(x,t) = \sum_{\wp} \upsilon(\Pi, x, t) S_A(\pi)$ and $\xi_B(x,t) = \sum_{\wp} \upsilon(\Pi, x, t) S_B(\pi)$. Similarly, the difficulty functions on the tested populations $\wp_A$ and $\wp_B$ become:

$$\zeta_A(x) = \sum_{\Xi} \xi_A(x,t).M(t) = \mathrm{E}_{\Xi,\wp_A}[\upsilon(\Pi, x, T)]$$

and

$$\zeta_B(x) = \sum_{\Xi} \xi_B(x,t).M(t) = \mathrm{E}_{\Xi,\wp_B}[\upsilon(\Pi, x, T)],$$

respectively.

Applying EL and LM results to the population of tested versions requires *conditional independence* between coincident failures, as in (4). In these models conditional independence follows 'naturally' from the fact that the versions are drawn at random (i.e. independently with replacement) from the populations. Once the versions are tested the urn model, see [4], is not applicable any more. The pairs are chosen before the testing and the versions in the pair evolve together affected by the chosen test suite(s). In the next section we check if and for which testing regimes the conditional independence still holds. Intuitively, we expect to see some differences between the testing regimes.

3.1. Testing with test suites independently drawn from the same population

3.1.1. Versions independently drawn from same population

Consider that a *pair* of programs is tested with two independently generated test suites. We still assume that the detection and the fault fixing are perfect, that is given a fault is activated by a demand in the test suite the failure is detected by the oracle and the causing fault is successfully removed. These assumptions limit the uncertainty about the effect of the tests on the tested versions: now this is limited to the revealing power of the test suite only. Consequently a given *test suite* represents a *single test*. The effect of the testing upon the failure of a program is to change its score function according to whether the program's ability to execute *x* correctly has changed as a result of the test: in this case, if *x* originally lay within a fault that has been executed in the test, then this fault will now have been removed and *x* will have been executed correctly.

Consider the probability of simultaneous failure of two randomly selected versions, Π_1 and Π_2, tested with randomly selected tests, T_1 and T_2, on a particular demand *x*:

$$P(\textit{both versions fail on } x) = \mathrm{E}_{\wp,\Xi}[\upsilon(\Pi_1, x, T_1)\upsilon(\Pi_2, x, T_2)] = \sum_{\wp}\sum_{\wp}\sum_{\Xi}\sum_{\Xi} \upsilon(\pi_1, x, t_1)\upsilon(\pi_2, x, t_2) S(\pi_1) S(\pi_2) M(t_1) M(t_2) \quad (15)$$

This can be simplified (see Appendix for details):

$$P(\textit{both versions fail on } x) = \mathrm{E}_{\wp,\Xi}[\upsilon(\Pi_1, x, T_1)\upsilon(\Pi_2, x, T_2)] = \zeta(x)\zeta(x) \quad (16)$$

Thus when versions are selected and tested in this way, they will exhibit conditional independence of failure for each demand *x*. That is, if there is conditional failure independence *before* testing, there will be conditional

independence *after* testing, so long as the test suites are themselves independent.

3.1.2. Versions independently drawn from different populations (forced design diversity)

By a similar argument to that above:

P(both versions fail on x) =

$$\underset{\wp,\Xi}{\mathrm{E}}\left[\upsilon(\Pi_1|\Pi_1 \in A, x, T_1)\upsilon(\Pi_2|\Pi_2 \in B, x, T_2)\right] = \zeta_A(x)\zeta_B(x), \quad (17)$$

Once again, there is conditional independence between version failures after testing.

3.2. Testing with test suites independently drawn from different populations (forced testing diversity)

Here we assume that two procedures for generating test suites exists, TA and TB, which we model by defining two probabilistic measures, $M_{TA}(\bullet)$ and $M_{TB}(\bullet)$, respectively, on the set of possible set suites, Ξ.

3.2.1. Versions independently drawn from same population

Inference similar to (16) leads us to the following expression of the probability of system failure:

P(both versions fail on x) =

$$\underset{\wp,\Xi}{\mathrm{E}}\left[\upsilon(\Pi_1, x, T_1)\upsilon(\Pi_2, x, T_2)\right] = \zeta_{TA}(x)\zeta_{TB}(x). \quad (18)$$

Again, there is conditional independence of version failures after testing.

3.2.2. Versions independently drawn from different populations (forced design diversity)

P(both versions fail on x) =

$$\underset{\wp,\Xi}{\mathrm{E}}\left[\upsilon(\Pi_1|\Pi_1 \in A, x, T_1 \in TA)\upsilon(\Pi_2|\Pi_2 \in B, x, T_2 \in TB)\right] = \zeta_{A,TA}(x)\zeta_{B,TB}(x), \quad (19)$$

Again, there is conditional independence between version failures after testing.

To summarise these four results: if the versions are tested on independently chosen test suites, the conditional independence is preserved after the testing, no matter whether diversity is employed in development only or in the selection of the test suites as well. We now consider the effect of testing the versions on the *same* test suite.

3.3. Testing with the same test suite

First we consider the case where the versions are selected independently from the same population (see Appendix for details):

P(both versions fail on x) =

$$\underset{\wp,\Xi}{\mathrm{E}}\left[\upsilon(\Pi_1, x, T)\upsilon(\Pi_2, x, T)\right] = \sum_{\Xi}\left(\xi(x,t)\right)^2 M(t) = \quad (20)$$

$$\left(\zeta(x)\right)^2 + \underset{\Xi}{Var}\left(\xi(x,T)\right) \geq \left(\zeta(x)\right)^2.$$

In the case where the versions are selected independently from *different* populations – forced diversity - the probability of coincident failure is:

P(both versions fail on x) =

$$\underset{\wp,\Xi}{\mathrm{E}}\left[\upsilon(\Pi_1, x, T)\upsilon(\Pi_2, x, T)\right] = \sum_{\Xi}\xi_A(x,t)\xi_B(x,t)M(t) = \zeta_A(x)\zeta_B(x) + \underset{\Xi}{Cov}\left(\xi_A(x,T), \xi_B(x,T)\right). \quad (21)$$

These results, (20) and (21), show that, although the versions fail conditionally independently on demand x *before* testing, this will generally not be true *after* testing. Essentially, the use of a common test suite has induced dependence in their failure behaviour.

For the independence of version failures to remain true after testing, it would be sufficient to have a constant efficiency for each test suite t, i.e. $\xi(x,t) = const$, (this is also *necessary* without forced diversity). Clearly, this is unrealistic.

(20) and (21) are important because they preclude using the EL and LM models (which assume conditional independence of failures on each demand *x)* once a two channel system is expected to be tested with the same test suite, which appears to be a common practice. Acceptance testing for fault-tolerant software, for instance, is based on the same test suite and so is the integration/system testing for such systems.

(20) asserts that testing both versions on the same suite implies on average that an (incorrect) assumption of conditional independence will be too optimistic.

(21) seems to salvage the assumption of conditional independence: in theory the covariance, $\underset{\Xi}{Cov}\left(\xi_A(x,T), \xi_B(x,T)\right)$ could even be negative, i.e. in some cases assuming conditional independence will be conservative. It is unclear, however, how realistic in practice it is for $\underset{\Xi}{Cov}\left(\xi_A(x,T), \xi_B(x,T)\right)$ to be negative, given $\xi_A(x,T)$ and $\xi_B(x,T)$ represent the efficiency of the same population of test suites on the (same) population of programs.

3.4. Marginal probability of system failure

All of the foregoing concerns what happens on one particular demand *x*. In practice, we are interested in the marginal probability of system failure (i.e. on a demand selected randomly via the operational demand profile). We consider the cases of unforced and forced version selection (design) separately.

3.4.1. Versions independently drawn from the same population

(16) and (20) show what happens conditionally on a particular demand, *x*: essentially, in each case the use of a common test suite increases the probability of simultaneous failure of the versions compared with the case when both versions are tested independently. Since this is true for each x, it follows that the use of a common test suite increases the *marginal* probability of system failure (i.e. on a *randomly chosen X*). More precisely:

$$P(\textit{both versions fail on } X \mid \text{independent test suites}) = \sum_F \left[\left(\zeta(x)\right)^2\right] Q(x) = \mathrm{E}\left[\left(\Theta_T\right)^2\right] = \left(\mathrm{E}[\Theta_T]\right)^2 + \underset{F}{Var}\left(\Theta_T\right), \quad (22)$$

$$P(\textit{both versions fail on } X \mid \text{same test suite}) \sum_F \left[\left(\left(\zeta(x)\right)^2\right) + \underset{\Xi}{Var}\left(\xi(T,x)\right)\right] Q(x) = \mathrm{E}\left[\left(\Theta_T\right)^2\right] + \sum_F \underset{\Xi}{Var}\left(\xi(T,x)\right) Q(x) = \quad (23)$$

$$\left(\mathrm{E}\left(\Theta_T\right)\right)^2 + \underset{F}{Var}\left(\Theta_T\right) + \sum_F \underset{\Xi}{Var}\left(\xi(T,x)\right) Q(x)$$

$$\geq P(\textit{both versions fail on } X \mid \text{independent test suites})$$

where $\Theta_T \equiv \zeta(X)$, by analogy with [1], is a random variable representing the proportion of tested versions failing on a randomly selected demand *X*. All this is intuitively plausible, since even though the programs started out as 'independent objects', the common testing they have undergone will have made them 'more alike'. Whilst it is obvious that on average the chance of simultaneous failures of two versions will be increased by their being tested on the same test suite, rather than on different suites, it is less obvious that the magnitude of the difference depends upon the variance term here.

This variance, $\underset{\Xi}{Var}\left(\xi(T,x)\right)$, is a non-negative number which can be substantial with a maximal value of 0.25 in the case $\zeta(x) = 0.5$ and $\xi(T,x)$ taking on values either 0 or 1 and nothing in between for various test suites. The point here is not to argue that having a large variance is likely, rather that if the effectiveness of the testing varies between test suites, which is plausible, then testing the versions together will make a two-channel system *on average* less reliable than if the two versions are subjected to the same 'amount of testing' independently, and that this difference *may be* substantial. The minimum of the variance is of course 0, but this is, as explained above, a very special case – when the testing makes $\xi(T,x) = const$, which is not realistic.

Note, however, that none of the above means that we should in practice necessarily prefer to test with different test suites. The question of what is optimal will involve a cost-benefit trade-off. The cost will be affected by the cost of generating the test suites and the cost of testing the versions on these. Two extreme scenarios can be identified for which different testing strategies may be optimal.

If the test generation is very expensive while running the tests is very cheap, testing versions independently on different test suites may not make much sense. The reliability improvements from testing versions on independently generated tests may be too small, while the benefits of removing more faults in versions too great to be missed, even at the expense of making the versions more alike. In other words, we can run twice as long a test (merging the two generated test suites) on each of the versions at the same cost as running each version with only one of the tests. Clearly, with the longer test not only the individual reliability of the versions is going to be better but so is the system reliability. The only way for the shorter tests to produce the same system reliability is for the second half of testing (the second test suite) to be completely ineffective, finding no faults. This would be possible, for instance, if the two test suites were identical (or the demands are perturbed) and becomes increasingly plausible as the individual reliability increases (the law of diminishing returns is operating). Testing versions together with a test twice longer, therefore, may be better than testing them individually on independently generated shorter test suites. The reason, of course, is the extreme assumption that the costs of running the tests are negligible.

At the other extreme is the case where the cost of the testing is mainly affected by the cost of running the tests, i.e. the cost of generating the test suites is low but running the tests is expensive. This seems to be a more realistic scenario than the previous one. Under this scenario it seems reasonable to limit the effort to testing the versions individually only, i.e. with different test suites. Our results are directly applicable in this case were we can generate many test suites but can afford to test each version on just one suite.

Finally, the most realistic scenario is, of course, when both the test generation and running of tests are expensive. Our models do not address this situation and the choice of the most cost-effective way of testing must be based on taking into account the detailed costs of generating and executing the tests. Simulation has been used to

investigate how the reliabilities of the versions and of the system improve as a function of testing effort [5].

3.4.2. Versions independently drawn from different populations (forced design diversity)

In case of *forced diversity* the probability of system failure depends on the covariance terms $\underset{\Xi}{Cov}(\xi_A(x,t),\xi_B(x,t))$, which may be positive or negative. Thus, in principle, the system tested with the same test suite can be more reliable than if the versions were tested individually. More precisely:

P(*both versions fail on X* | independent test suites) =

$$= \sum_F [((\zeta_A(x)\zeta_B(x)))]Q(x) = \mathrm{E}[(\Theta_{A(T)}\Theta_{B(T)})] = \quad (24)$$

$$\mathrm{E}[\Theta_{A(T)}]\mathrm{E}[\Theta_{B(T)}] + \underset{F}{Cov}(\Theta_{A(T)},\Theta_{B(T)}),$$

where $\Theta_{A(T)} \equiv \zeta_A(X)$ and $\Theta_{B(T)} \equiv \zeta_B(X)$, by analogy with [2], are random variables representing the proportion of tested versions failing on X for methodologies A and B, respectively. Similarly,

P(*both versions fail on X* | tested on same test suite) =

$$\sum_F \left[((\zeta_A(x)\zeta_B(x))) + \underset{\Xi}{Cov}(\xi_A(T,x),\xi_B(T,x)) \right] Q(x) =$$

$$\mathrm{E}(\Theta_{A(T)})\mathrm{E}(\Theta_{B(T)}) + \underset{F}{Cov}(\Theta_{A(T)},\Theta_{B(T)}) \quad (25)$$

$$+ \sum_F \underset{\Xi}{Cov}(\xi_A(T,x),\xi_B(T,x))Q(x).$$

Comparing (24) and (25) does not allow one to conclude which of the testing regimes will produce on average a more reliable two-version system. The difference between (24) and (25), is the term $\sum_F \underset{\Xi}{Cov}(\xi_A(T,x),\xi_B(T,x))Q(x)$. This is a sum of covariances each of which can be a positive or a negative number. It is not clear what the balance between the positive and the negative terms will be. In the cases where this expression is positive, testing the versions individually will produce a more reliable pair, as in the case without forced diversity. If $\sum_F \underset{\Xi}{Cov}(\xi_A(T,x),\xi_B(T,x))Q(x)$ is negative, however, then testing the versions on the same test suite will produce a more reliable pair, which is counterintuitive because it means that by testing more cheaply (only one testing suite will need to be generated) a more reliable system can be delivered.

We can analyse specific cases under additional assumptions, e.g. assuming disjoint failure regions as in [6], [7], but the general constraints under which $\underset{\Xi}{Cov}(\xi_A(x,t),\xi_B(x,t)) \le 0$ are currently not known. More refined analysis is needed to reveal the constraints which, if imposed on the generation of the tests suites, will make $\underset{\Xi}{Cov}(\xi_A(x,t),\xi_B(x,t))$ more likely to be positive or negative.

4. Other testing

4.1. Testing with imperfect oracle and imperfect fault-fixing

If the oracle is imperfect and/or fault-fixing is imperfect, there is extra uncertainty involved in the testing process and the kind of modelling presented above becomes more difficult. The best we can do is find some bounds for the system probabilities of failure.

Assume, for simplicity, that introducing new faults during testing is impossible. Such an assumption is not unusual in software reliability modelling: for example, most reliability growth models rely on such an assumption. Clearly, under this assumption a tested version will have scores on individual demands no better than if tested with perfect oracle/fixing. Thus, the results from the previous section (15-25) can be used as lower bounds on the probability of system failure. Equally, the scores will be no worse than the scores of the untested version which thus forms a natural *upper* bound (though not a very useful one) on the probability of system failure.

4.2. Back-to-back testing

Back-to-back testing is a special case of testing the versions on the same test suite. The failures are detected if a mismatch occurs between the outputs of the versions. This is the main practical advantage of back-to-back testing: it does not require an oracle to judge whether the output of a software version is a failure or success. If at least one version (in a fault-tolerant software system) succeeds on a demand then detection of any failures of other versions is guaranteed. If, however, all versions fail coincidentally on a demand then successful detection is not guaranteed: there is a possibility that all versions fail in exactly the same way in which case there will be no mismatch of the outcomes.

Our model does not allow us to analyse the most general case where there is only a *possibility* of detection. The most we can do, again, is obtain bounds for the evolving system reliability based on the most optimistic and most pessimistic scenarios.

If we assume *optimistically* that coincident failures are *never* identical we can guarantee failure detection. The results are then the same as those involving a perfect oracle obtained in section 3.

The most *pessimistic* assumption about failure detection, on the other hand, assumes that *all* coincident

failures are identical. Such failures are undetectable. In this worst-case scenario, back-to-back testing does not improve system reliability at all – it only improves the reliability of the individual versions on demands which have no effect on system reliability. Essentially the version reliability improvements are exactly matched by worsening diversity between versions as testing proceeds. In the limit (after exhaustive testing), the versions would fail identically and the system behave exactly as each version does.

5. Conclusion

The work reported here is part of an ongoing effort to acquire a more formal understanding of the effect of diversity upon system dependability. At one level, diversity is 'clearly a good thing' – this is reflected in its ubiquity in human affairs. On the other hand, the precise mechanisms that underpin its efficacy are still not well understood. However, it is becoming clear that simple intuition can sometimes be misleading hence the need for careful formal modelling.

The earlier models of Eckhardt and Lee, and Littlewood and Miller, were attempts to provide a formal probabilistic framework for the use of design diversity in achieving system dependability. Essentially these models took a snap-shot view of the system. The present work extends these models to represent the *evolution* of system dependability that takes place as a result of testing.

The results presented in this paper are still more in the area of conceptual answers than advice on how to apply diversity to achieve reliability in practice. The results we have presented – e.g. that testing versions on independently generated suites has a greater potential to improve system reliability - are not surprising. Essentially, the results simply confirm previous intuition to 'make the versions as independent as possible'. Thus *using the same test suite* means introducing a 'channel' of dependence between the failures of the versions, making the two tested versions more likely to fail together on a specific demand than under the assumption of their failing independently.

On the other hand, the results give detailed insight into the way diversity impacts upon system dependability. They do this via generalisations of the earlier models, i.e. through the use of moments of 'difficulty functions'. It remains a moot point whether these variances and covariances that determine the efficacy of diversity in design and testing can be estimated in practice. This remains an open research issue.

The formalism in modelling the evolution of diverse software presented here seems applicable to modelling any kind of commonality, not only those arising from testing on the same test suite. There are many other ways in which commonalities can be introduced into otherwise independent development of several versions. For instance, if an ambiguity is discovered by one of the teams, and a common clarification is sent to all development teams, this can conceptually be modelled as running the same 'test suite' against all versions. The difference in this case, compared with the description given in section 3, will be that the common test suite is not generated to cover the whole demand space and be representative of the operational usage of the versions, but instead will affect a (possibly small) sub-set of the demand space. The effect of this propagation of the common knowledge, however, may be the same – reducing diversity of the versions.

Another instance of the 'same test suite' approach, which may be useful to explore, is the representation of common mistakes (e.g. giving *incorrect* instructions to all teams about how to resolve ambiguities in the specification). The difference in this case, compared with the description in section 3, is that the 'common test' will result in setting the scores of all demands affected to 1 (i.e. make versions produce incorrect results) instead of fixing the mistakes. Detailed modelling is left for the future.

In practical software development a combination of different activities is utilised which introduce sources of dependence between the channels. We intend to study the effect of applying more than one activity to the diverse channels and the interplay between their individual characteristics (e.g. efficacy) and mutual diversity which may reveal practical ways of enforcing useful diversity between the final software products.

Acknowledgement

This work was partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under the 'Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems (DIRC)', by DISPO-2 (DIverse Software PrOject) Project, funded by British Energy Generation Ltd and BNFL Magnox Generation under Contract No. PP/40030532f and by the European Commission under the BIS-21 project, Centre of Excellence for Education, Science and Technology, Contract No. ICA1-2000-70016.

The authors would like to thank Bob Yates and Lorenzo Strigini for the helpful comments on earlier drafts of this paper.

References

1. Eckhardt, D.E. and L.D. Lee, *A theoretical basis for the analysis of multiversion software subject to coincident errors.* IEEE Transactions on Software Engineering, 1985. **SE-11**(12): p. 1511-1517.
2. Littlewood, B. and D.R. Miller, *Conceptual Modelling of Coincident Failures in Multi-Version Software.* IEEE

Transactions on Software Engineering, 1989. **SE-15**(12): p. 1596-1614.
3. Littlewood, B. and D. Wright, *Some conservative stopping rules for the operational testing of safety-critical software.* IEEE Transactions on Software Engineering, 1997. **23**(11): p. 673-683.
4. Miller, K.W., L.J. Morell, et al., *Estimating the Probability of Failure When Testing Reveals No Failures.* IEEE Transactions on Software Engineering, 1992. **18**(1): p. 33-43.
5. Djambazov, K.B. and P. Popov. *The effects of testing on the reliability of single version and 1-out-of-2 software*. in *6th International Symposium on Software Reliability Engineering, ISSRE'95*. 1995. Toulouse: IEEE Computer Society Press. p. 219-228.
6. Littlewood, B., P. Popov, et al., *Modelling the effects of combining diverse software fault removal techniques.* IEEE Transactions on Software Engineering, 2000. **SE-26**(12): p. 1157-1167.
7. Popov, P. and L. Strigini. *The Reliability of Diverse Systems: a Contribution using Modelling of the Fault Creation Process*. in *DSN 2001 - The International Conference on Dependable Systems and Networks*. 2001. Goteborg, Sweden. p. 5-14.

Appendix

The derivation of expression (16) is detailed below:

$$P(\text{both versions fail on } x) = \mathop{\mathrm{E}}_{\wp,\Xi}\left[\upsilon(\Pi_1,x,T_1)\upsilon(\Pi_2,x,T_2)\right] =$$

$$\sum_{\wp}\left(\sum_{\wp}\left(\sum_{\Xi}\sum_{\Xi}\upsilon(\pi_1,x,t_1)\upsilon(\pi_2,x,t_2)M(t_1)M(t_2)\right)S(\pi_1)\right)S(\pi_2) =$$

$$\sum_{\wp}\left(\sum_{\wp}\sum_{\Xi}\upsilon(\pi_2,x,t_2)\left(\sum_{\Xi}\upsilon(\pi_1,x,t_1)M(t_1)\right)M(t_2)S(\pi_1)\right)S(\pi_2) =$$

$$\sum_{\wp}\left(\sum_{\wp}\left(\sum_{\Xi}\upsilon(\pi_2,x,t_2)\varsigma(\pi_1,x)M(t_2)\right)S(\pi_1)\right)S(\pi_2) =$$

$$\sum_{\wp}\left(\sum_{\wp}\varsigma(\pi_1,x)\left(\sum_{\Xi}\upsilon(\pi_2,x,t_2)M(t_2)\right)S(\pi_1)\right)S(\pi_2) =$$

$$\sum_{\wp}\left(\sum_{\wp}\varsigma(\pi_1,x)\varsigma(\pi_2,x)S(\pi_1)\right)S(\pi_2) =$$

$$\sum_{\wp}\varsigma(\pi_2,x)\left(\sum_{\wp}\varsigma(\pi_1,x)S(\pi_1)\right)S(\pi_2) =$$

$$\sum_{\wp}\varsigma(\pi_2,x)\zeta(x)S(\pi_2) = \zeta(x)\sum_{\wp}\varsigma(\pi_2,x)S(\pi_2) = \zeta(x)\zeta(x).$$

Expression (20) is derived as follows:

$$P(\text{both versions fail on } x) = \mathop{\mathrm{E}}_{\wp,\Xi}\left[\upsilon(\Pi_1,x,T)\upsilon(\Pi_2,x,T)\right] =$$

$$\sum_{\Xi}\left(\sum_{\wp}\left(\sum_{\wp}\upsilon(\pi_1,x,t)\upsilon(\pi_2,x,t)S(\pi_1)\right)S(\pi_2)\right)M(t) =$$

$$\sum_{\Xi}\left(\sum_{\wp}\upsilon(\pi_2,x,t)\left(\sum_{\wp}\upsilon(\pi_1,x,t)S(\pi_1)\right)S(\pi_2)\right)M(t) =$$

$$\sum_{\Xi}\left(\sum_{\wp}\upsilon(\pi_2,x,t)\xi(x,t)S(\pi_2)\right)M(t) =$$

$$\sum_{\Xi}\xi(x,t)\left(\sum_{\wp}\upsilon(\pi_2,x,t)S(\pi_2)\right)M(t) =$$

$$\sum_{\Xi}\xi(x,t)\xi(x,t)M(t) = \sum_{\Xi}\left(\xi(x,t)\right)^2 M(t) =$$

$$\left(\zeta(x)\right)^2 + \mathop{Var}_{\Xi}\left(\xi(x,T)\right) \geq \left(\zeta(x)\right)^2.$$