

Choosing Effective Methods for Design Diversity - how to progress from intuition to science

Peter Popov¹, Lorenzo Strigini¹,
Alexander Romanovsky²

¹Centre for Software Reliability, City University, Northampton Square,
London EC1V OHB, U.K.
{ptp, strigini}@csr.city.ac.uk

²Centre for Software Reliability, University of Newcastle-upon-Tyne,
Newcastle upon Tyne NE1 7RU, U.K.
Alexander.Romanovsky@newcastle.ac.uk

(Presented at SAFECOMP'99. Appears in Lecture Notes in Computer Science, Vol. 1698.

This version corresponds to Version 1.0, 11 February 1999, of the DISPO Technical Report of the same title)

Abstract. Design diversity is a popular defence against design faults in safety critical systems. Design diversity is at times pursued by simply isolating the development teams of the different versions, but it is presumably better to "force" diversity, by appropriate prescriptions to the teams. There are many ways of forcing diversity. Yet, managers who have to choose a cost-effective combination of these have little guidance except their own intuition. We argue the need for more scientifically based recommendations, and outline the problems with producing them. We focus on what we think is the standard basis for most recommendations: the belief that, in order to produce *failure* diversity among versions, project decisions should aim at causing "diversity" among the *faults* in the versions. We attempt to clarify what these beliefs mean, in which cases they may be justified and how they can be checked or disproved experimentally.

1 Introduction

This paper is a preliminary discussion of the main hitherto un-addressed questions in achieving effective design diversity, i.e., producing diverse-redundant systems with low probability of common-mode failures of the channels.

Developers of critical systems often employ diversity between redundant channels. Redundancy protects the system against physical failures of the individual channels, but leaves it vulnerable to design faults which, if repeated in them all, can

cause common-mode failures. So, in applications such as nuclear plant protection it is common to employ parallel, diverse channels. Each channel separately inputs and processes plant data and can trigger a safe shut-down if it detects indications of unsafe conditions. Two current trends are increasing the interest for design diversity: increased reliance on off-the-shelf products, which may lack complete documentation of quality development procedures, and the practical disappearance of non-software based alternatives (e.g., non-smart sensors) for many functions.

For software, design diversity is sought by having two or more separate teams develop variants (often called *versions*) of a program. It is hoped that, if one version fails, the other[s], being internally different, will not fail at the same time: if they contain bugs, these will not cause failures in exactly the same circumstances in all versions. The versions must exhibit the same functional (externally visible) behaviour. The two or more versions are then run in a redundant configuration, so that failures in a subset of the versions may be masked or at least detected. More refined arrangements are possible, e.g. with some version only performing a monitoring or auditing function on others which have active control functions [1, 2]. Other benefits are also sought from implementing multiple versions, e.g., "back-to-back" testing provides a cheap, though imperfect, oracle for automated testing.

An important problem with design diversity (as with most other techniques for reducing or tolerating design faults) is that the reliability gain that it produces is difficult to evaluate. We know that one cannot assume diverse versions to fail independently, and all other techniques for assessing high levels of reliability are no less problematic for multiple-version than for ordinary software. For a summary of research results on this problem readers can refer to [3-5].

The other important question is how best to achieve effective diversity, i.e., a low probability that the versions will fail together. A project manager can indirectly control this by various decisions. To preserve diversity, the teams developing the versions are typically not allowed to exchange information about the development. Considering that people engaged in similar activities often make similar mistakes, they may also be given explicit directives for diversifying the internal structures of their products (e.g., using different algorithms). However, how do we know that these decisions will actually improve the delivered multi-version product?

The existing literature, and even standard documents, contain lists of such decisions that a design manager can apply to pursue diversity, which can be seen as "common-sense" advice (e.g., [6] gives developer-oriented advice, [7, 8] give customer-oriented requirements). For brevity, we shall call them "DSDs", for "diversity-seeking decisions". A complete list of plausible DSDs would span the whole development process, from team selection, to using different development environments, different tools and languages at every level of specification, design and coding, implementing each function with different algorithms, applying different V&V methods, etc. Some DSDs (like choice of algorithms) will be specific to an individual product.

But how can a project manager choose from such a "shopping list"? One may think that the more DSDs are applied, the better. But most of them have a cost: duplication of activities, added co-ordination effort, need for staff with specific

skills. How many DSDs are enough for the desired level of assurance against design faults, or what is a cost-effective set of DSDs? There is currently no scientific answer to this question. We are not even sure that the advantages from various DSDs add up. We could think, for instance, that a DSD (say, specifying diverse algorithms for the various versions) produces benefits by giving a team a "scrambled" version of the problem seen by another team, so that they are not likely to make the same mistakes. However, perhaps there is a point beyond which further "scrambling" produces no further advantage: the problems seen are already as different as they can be. Then, applying a second DSD (say, using very different design methods for the various versions), possibly just as effective as the first one when used alone, would not give any additional advantage when used in combination with it.

In short, to decide which DSDs should be applied in a given project one needs to answer questions about their effectiveness (individually and in combinations). Then, a rational choice would become possible, considering known costs and other practical constraints. But the effectiveness of the various, plausibly useful DSDs is unknown. Instead, a project manager or system-level designer now has little to rely on except intuition, guided by personal experience. Experience is a poor guide for drawing general laws on how to avoid problems that are very rare in the first place; and intuition has been shown repeatedly to fail on these matters: the issues with diversity are subtle, and difficult even to define properly. For instance, some developers maintained that design-diverse channels would obviously fail independently, until this was proven wrong by theory and experiments alike. Even now, the assumption is often made that "functional diversity" (in which the channels perform similar system-level functions, but using different input data, different actuators and generally different techniques) guarantees independence of failures, a view that is refuted in [9].

In this paper, we examine how the technical community can gain better understanding of how useful diversity is generated, and thus projects can better choose among DSDs. The first step is to define more clearly the questions to be asked and recall what is known about the answers (Section 2). Next, we will examine what evidence can be produced of the efficacy of a DSD (Section 3). In Section 4, we detail the most common form of intuitive argument in favour of a DSD, which is based on its presumed efficacy in causing "diverse" development errors and/or "diverse" defects in the versions. In Section 5, we try and formalise these arguments via models that could be used in practice: we describe situations in which the empirical data could give a strong scientific basis for trusting a DSD. Section 6 contains our conclusions.

2 What Is Useful Diversity and What Is Known About It

2.1 Essential Terminology

We first need to define a few terms. We will say that a version (or a system) *fails* when its behaviour deviates from what it should be. We will say that a failure is

caused by a *defect* or *fault* in the code. Any human error that caused the defect to be in the code as executed will be called a *mistake* (although this term is often given a more specialised meaning in psychology [10]). A mistake could be, for instance, mis-stating or omitting to state a required function in a requirement document; failing to notice a defect in a specification document during an inspection; accidentally using the wrong variable as target of an assignment statement, during coding; omitting to test under a certain condition that would reveal a fault.

To keep things simple, we will always refer to the simplest diverse-redundant configuration: two versions in a "1-out-of-2" configuration, in which proper operation of one version is sufficient for the whole system to function properly.

Most discussions about diversity use the terms "diversity" and "independence" in a rather informal way. We need to clarify their meaning, if we wish to learn more about them by scientific methods. First of all, the term "diversity" may designate several concepts (Fig. 1). DSDs produce "process diversity". They presumably cause the versions to be visibly different in their structure and internal operation ("product diversity"). They also cause -one hopes- the versions to be less likely to contain identical defects than if the DSD were not employed in the first place ("fault diversity"). And finally, if successful, they reduce the probability that the versions will fail in the same way on the same demands ("failure diversity"). Failure diversity is the actual goal of DSDs. All the rest are means to this end, and without more analysis we cannot even be sure that they are necessary means, rather than unnecessary side-effects.

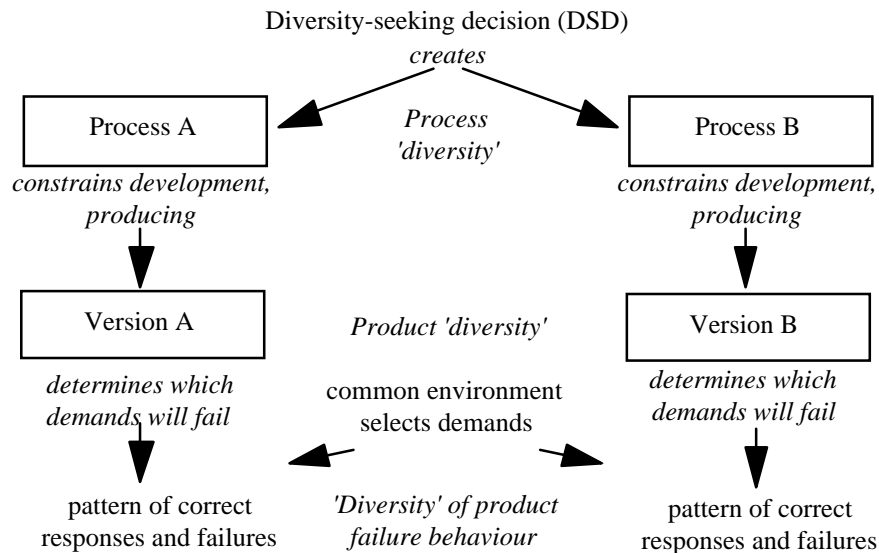


Fig. 1. The causal links from DSDs to failure diversity

"Independence" of failures between two versions means simply that the probability of the two failing together on a demand is the product of their individual probabilities. Many practitioners and some standards [7, 8] advocate (or prescribe) that versions be developed "independently" to achieve effective diversity. Strict separation between the teams seems the solution. However, modelling has shown [3] that even if we can achieve this perfect "causal independence", we should expect, on average, positive correlation between version failures; while "forced diversity" may in theory achieve lower correlation between version failures, including independence or even negative correlation. "Forced diversity" means, in the terminology of this paper, applying *some* DSD[s]; and those demanding "independent development" would usually agree that it is desirable. However, DSDs may actually impose more common constraints on the developments, and generally mean that the developments are not "statistically independent" (as one attempts to achieve negative correlation between some characteristics that are deemed important). So, the word "independence" is applied to development to mean a form of statistical dependence; while it is applied to failures in the statistical sense.

2.2 Modelling Diversity: the Need for Probabilities

The explanation of the modelling results quoted above [3, 4] is that one can see the development of a version as an uncertain process. The versions actually produced are, in practice, picked (independently, by assumption, when modelling perfectly independent developments) from a distribution of versions that *could have been*

produced. The problem that they have to solve is the same for them all. Presumably, some of the demands pose a more difficult problem than others, in the specific sense that versions are more likely to fail on the former than on the latter. The mathematics then shows that this uneven distribution of "difficulty", common for all versions, causes, on average, positive correlation among their failures, when averaged over the profile of demands to which the versions will be subjected. However, by forcing diversity, i.e., by DSDs that cause process diversity, one might cause two teams to encounter different distributions of "difficulty": the demands that are especially difficult for the team using one process may be the easier ones for a team using the other process. This of course decreases the risk of common failures.

The conclusion is that forcing diversity - by adopting a DSD - is certainly beneficial (on average) if the DSD creates two process variants that 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. Consider for instance an organisation that uses two languages, A and B. Suppose that experience has shown that the choice of language does not seriously affect program reliability. Having to develop a two-version system, the right choice is then to develop one version in language A and the other in B. However, if this organisation had found language A to produce generally more reliable programs than B, deciding whether the two-version system should be an A-A or an A-B system would require far more information (see [3] for details).

This leads to another consideration about any search for effective DSDs: effectiveness is judged in terms of future results, but these will vary among projects that apply a given DSD. So, effectiveness must be stated in terms of probabilities. As for measures of effectiveness, many choices are possible. A DSD could be judged effective enough if it reduces by 1/2 the probability of common failures among two versions (in a project of a certain type); or if it reduces by 1/2 the probability of common failures being more likely than 10^{-3} ; and so on. The choice depends on one's requirements. The problem in arriving at such statements is one of *prediction*. As usual in engineering, one can only decide how to *achieve* certain results by *assessing* the effects that the means proposed will have on the products "in general". In other words, by deciding how much the use of a certain DSD should increase one's confidence in the product delivered, before the product is actually built. So, the prediction is about the statistical effects of the DSD on the products that *may* be developed for a given requirement, without the benefit of information about the individual product of interest. The fact that precise predictions are probably infeasible should not discourage us: for decision-making it is usually sufficient to know whether a certain decision is "substantially better than" (or even simply "at least as good as") another one.

2.3 Empirical Evidence

We know very little about the general efficacy of any specific DSD. Experiments [2, 1] have seldom been analysed from this viewpoint (attempts are presented in [11,

12], and some interesting considerations in [13]). In any case, they only provide anecdotal evidence: each experiment only developed multiple versions of one program, leaving open the doubt whether a DSD that appeared beneficial would be so when developing another program. In addition, most experiments have developed toy programs in artificial, non-industrial environments. Industrial examples exist, and companies publicise the DSDs they use. For instance, the CBI railway signalling system uses C and assembly language for diverse channels [14]; Airbus has used a specialised process control specification language with a code generator side by side with a conventionally developed version [15]; software in the ELEKTRA railway signalling system [16] has a "conventional" primary channel side by side with a rule-based safety-monitoring channel. But published failure data from these products only indicate that they are very reliable. Reasoning about whether different DSDs (or none at all) would have produced much different reliability would require detailed analysis, which, though worth attempting, risks being inconclusive when based on a handful of failures or of defects. On the other hand, even if all channels in such products were perfectly fault-free, this would not mean that the DSDs used by these companies were useless (an argument occasionally heard is "if diversity has not prevented any accident, why not drop it and save the extra cost?"). The purpose of DSDs is to ensure that *even if* the versions contained faults (which developers try to make unlikely anyway, but cannot make impossible), these faults would be less likely to cause system failures. In practice, observing an association between a DSD and high system reliability will not be enough to argue that one caused the other, unless we understand the causal mechanisms by which the DSD promotes high reliability.

3 Demonstrating the Efficacy of a Development Decision

Suppose we wish to decide whether a certain DSD is likely to be effective enough to be employed in our next 2-channel system. For the sake of concreteness, imagine the DSD is the use of the Ada language in one version and of C in the other. To convince ourselves that a certain DSD is sufficiently effective for a certain use, we can in theory proceed in different ways:

1. since the only measure that matters is the reliability of 2-version systems, we could collect reliability data about many pairs of versions developed with the Ada-C DSD, and many pairs which did not employ it (i.e., used either Ada only or C only). Statistical analysis would allow us to filter out the effects of other factors (general quality and effort spent in the various projects, for instance). Clearly, this approach is not generally practical. Pairs of programs performing essentially the same functions exist, but they are produced under so many different conditions that the data filtering exercise would be close to hopeless, even if the organisations concerned could provide the data in comparable formats. Two-channel systems are rare; separate programs that perform similar functions in different systems may be subject to different demand profiles, so that failure data may not be comparable. For programs of very high quality, failures

might be so rare that we could not estimate the advantage brought by diversity.

We could produce multiple pairs via controlled experiments, but costs would limit us to toy programs or non-professional developers. To believe that whatever results we obtain extend to real industrial developments, we would need first to understand the cause-and-effect chains via which the DSD affects reliability, so well to believe that they are the same in the lab and in an industrial environment.

2. we could use the models described in [3, 4]: by estimating how "difficulty" varies among demands for versions developed with either Ada or C. However, "difficulty" is not defined here in terms of the likelihood that developers make mistakes when dealing with certain design problems, i.e., something of which we may have direct experience from the past. It is defined in terms of the probabilities that these mistakes cause failures on certain demands, weighted with the probabilities of those demands. To confidently evaluate our DSD, we would need hard empirical evidence. But to collect it, we would have the same problems indicated above, of scarcity of data, multiplied by the fact that we would need detailed reliability data for individual demands (which is impossible) or classes thereof (which is demanding), rather than for the whole operation of each program.
3. last, we could try and refine the common intuition that "a DSD in some sense causes 'fault diversity'; 'fault diversity' in some sense causes 'failure diversity'" (we are using "fault diversity" as shorthand for "tendency of versions to exhibit qualitatively different fault patterns"). If we can model these two cause-effect links, we could then try and demonstrate their existence separately. In particular, the "fault diversity" effect of a DSD may be relatively easy to measure. Practitioners have experience of the relative frequencies of different kinds of defects in different development processes. Besides, some DSDs make some types of fault impossible in one of the versions. Psychologists have studied human error [10], and their knowledge could be applied either to argue that our DSD should indeed cause "fault diversity", or to design economical experiments to check such conjectures.

There is less evidence for a link between fault diversity and failure diversity. Last but not least, what we wish to be able to demonstrate is that the two links of the chain, if both proven, combine to prove the property of interest, i.e. that the DSD causes failure diversity. For a start, this requires one to define "fault diversity" in a similar way for both links, and this definition to be such that "fault diversity" and its effect on failure diversity can be measured by feasible experiments. One also needs a mathematically tractable model of how they would combine.

This third, two-step method of demonstrating the efficacy of a DSD is appealing as it would allow greater reuse of knowledge than the other two. If we could show that a certain kind of 'fault diversity' has a useful effect on 'failure diversity' as a general rule, we could evaluate a DSD without expensive measurements on failures, as required by methods 1 and 2. We would only need evidence about the 'fault diversity' it causes, and even this could in some cases be based on general

psychological knowledge about the effects of different problem-solving constraints on human error. The rest of the paper is devoted to studying method 3 in more detail.

4 The Fault-Failure Chain: Informal Discussion

In this section, we describe difficulties with the intuitive notion that diversity "propagates" down the fault-failure chain. This appealing notion cannot be taken for granted. The similarity of the terms "failure diversity" and "fault diversity" is misleading. Failure diversity refers to "failing on different demands" (or "failing with different failure behaviours albeit on the same demand"), and specifically high failure diversity means a low probability that both version will fail equally on the same demand. Fault diversity is a subtler concept: a *tendency* of versions to exhibit qualitatively different faults. When we compare the faults in two programs, we can decide whether we think they are different from some viewpoint, e.g., because they seem to be caused by qualitatively different mistakes (a programmer's typo vs misinterpreting the specifications) or they cause different failure behaviours (e.g., a memory violation trap vs taking a wrong branch). However, we are now interested in how a DSD affects the *potential for* diversity among the *unknown* faults that *may* remain in a pair of versions when deployed.

Our problems in linking fault diversity to failure diversity arise from:

1. the difficulty of linking faults (defects in the code) to the specific demands on which they would cause failures;
2. the fact that the importance of a fault depends on the *probability* of those demands on which it causes failures.

About item 1, we notice that a fault can be identified in two ways (neither method guarantees unique identification - i.e., that all analysts will agree on the list of the faults in a given product- but we can neglect this difficulty for the time being):

- as a code defect, defined by its position in the code and its type. This presents a difficulty. Two defects in diverse versions, which use different variable identifiers, and possibly different languages, will hardly ever look identical, even when they would cause common failures. So, versions may always have "different" faults, and useful metrics of "how different" they are become difficult to define;
- as the set of demands on which the defect causes the version to fail (a "failure region" in the "demand space"). This is a less common view, but with it one can decide objectively whether two failure regions are disjoint, overlapping (and by how much) or coincident, and from this information define measures of diversity.

Unfortunately, there is no general, intuitive law linking failure regions to the defects that create them. Defects that appear similar either in type or location may never cause failures on the same demands, while defects that are different in appearance and caused by different mistakes may produce failures on the same demands.

Since two versions may be such that recognising faults as identical between the two may be impossible, and yet a pair of faults (one per version) that affected

overlapping sets of possible demands would of course cause system failures, "fault diversity" must somehow be referred to "types of faults", or "positions of faults" rather than to individual faults. Many schemes for fault classification are in use (e.g. [18]), so that statistics are available about the frequencies of the various types. Whether these data can be used for our investigation is yet to be seen.

Different considerations apply to diversity in the *location* of faults. If we can create a mapping between the parts of code that perform similar functions in the two versions, we can in principle measure the frequency with which a certain pair of processes (a DSD) creates defects in the same parts of the two versions. A lower frequency would be an indication of a more effective DSD. Intuitively, the reason for considering this form of "fault diversity" desirable is that defects that affect "corresponding" parts of the code in two versions are more likely to cause failures on the same demands than defects affecting "non-corresponding" parts. This cannot be taken for granted, but is plausible and can be studied empirically.

5 The Fault-Failure Chain: Special Cases

We have argued that if one chose a DSD at random from those that are commonly advocated as useful, one would be hard put to produce a convincing argument that it is indeed so. We now consider a related question: are there special (and plausible) circumstances that would demonstrate that a certain DSD is useful? We are not looking for an exemption from scientific rigour. Few useful engineering techniques have been found by working from first principles. More simply, engineers have demonstrated that a specific technique worked, and then adopted it. So, if researchers can state a set of verifiable, sufficient conditions for a DSD to be practically useful, they will have given practitioners a chance of confirming that a specific DSD is useful (or perhaps that it is not) in their circumstances, though no guarantee that they can reach such definitive judgements on all possible DSDs.

5.1 First Scenario: Relying on Experience of Mapping between Fault Classes and Failure Classes

Suppose that experience has shown that (for the class of software considered) the vast majority of failures are due to faults that affect (for each version) disjoint sets of demands, and these sets are the same for all versions. An example could be those faults that are due to misinterpreting the specified response for some particular class of demands. We could then define the set of all possible faults, $\{F_1, F_2, \dots, F_N\}$, and similarly index the sets of demands on which each fault causes failures (*failure regions*): $\{R_1, R_2, \dots, R_N\}$. Our DSD creates two development processes, A and B. These determine the probabilities of each fault being present in versions produced with either process: $P_A(F_1), P_A(F_2), \dots, P_A(F_N)$ and $P_B(F_1), \dots, P_B(F_N)$. The operational environment will determine the probabilities of demands that belong to the various regions: $P(R_1), P(R_2), \dots, P(R_N)$. If the two versions are developed "truly

independently" (i.e., if the only commonalities between the development efforts for these two versions are in the problem to be solved, which would affect *any* development of versions for this system, rather than being due to interactions between these two specific development efforts), the probability of failure is:

$$\begin{aligned} \sum_{i \in [1, N]} P_A(F_i) P(R_i) & \quad \text{for a single version produced with process A} \\ \sum_{i \in [1, N]} (P_A(F_i))^2 P(R_i) & \quad \text{for 2 versions, both produced with process A} \\ \sum_{i \in [1, N]} P_A(F_i) P_B(F_i) P(R_i) & \quad \text{for 2 versions, produced with processes A and B} \end{aligned}$$

It can be shown that if both processes give the same average reliability, an A-B system is better, on average, than either an A-A or a B-B system. If, say, process A gives the higher average reliability, then a B-B pair should be avoided, but the choice between an A-A and an A-B pair depends on the values of all the individual probabilities in the formulas. Since we would not know the detailed set $\{F_1, F_2, \dots, F_N\}$, we could not decide on this basis. However, we may well know the relative frequencies of different *classes* of faults observed with process A and with process B in the past, and thus have estimates for their relative frequencies in the current project. It is conceivable that we know that different *classes* of faults typically affect different sets of possible demands. For instance, perhaps initialisation defects and defects concerning the sequencing of concurrent activities tend to affect different kinds of demands. But even with this one-to-one mapping between defects in the code and subsets of the demand space, theorems in [5] show that predicting the overall failure rate requires the unknown parameters (probabilities of individual faults and of the demands that would trigger them) to satisfy special conditions. In conclusion, one cannot use this form of argument about the fault-failure chain without first proving by experiment several rather strong laws governing software development in the environment of interest.

5.2 Second Scenario: Relying on Probability of No Faults

Another possibility may present itself for organisations with a high-quality process: it may turn out that a given process (say, A) can be trusted with a certain probability (say, $1 - P_A(i)$) completely to avoid faults that would affect a given category of demands, i . In this case, a basis for dependability assessment could be the probability of having any failure at all during the lifetime of a two-version system. A certain probability of no faults of a given category is a lower bound on the probability of none of the failures that they might cause happening over the whole system lifetime (see [19] for a detailed discussion). So, for instance, in the simple scenario below the decision would clearly be that an A-B pair is to be preferred to an A-A pair, although process A produces on average better versions.

class of demands	probability of demand class	upper bound on prob. of failure		
		process A	process B	minimum
demand class 1	0.7	1%	10%	1%
demand class 2	0.3	5%	3%	3%
upper bound on probability of system failure:		0.022	0.079	0.016

5.3 Third Scenario: Diversity in the Positions of Faults

We now switch to the apparently more promising, alternative measure of "fault diversity" in terms of differences in the *positions* of the faults (cf end of Section 4). An interesting shortcut now appears possible for collecting empirical evidence of the efficacy of a DSD. The main intuitive basis for believing that defects affecting different parts of the code are likely to give low probability of common failure is the belief that these different parts of the code will often be invoked by different demands. It is then apparent that we could directly try and measure the effects of DSDs on the probabilities of non-null intersections between failure regions in different versions. The costs of experiments would still be high, but there is an interesting simplification compared to procedure 1 in section 3. If the failure regions in two versions have null intersection, the two versions will never fail together, *no matter* what distribution of demands they are subjected to. So, experimenters would not need to test the versions with realistic demand profiles, and be concerned that different profiles would invalidate the experimental results. An analysis of the "static" characteristics of failure regions would be substituted for a "dynamic" measure of failure probabilities. Such experiments could at least demonstrate the efficacy of a DSD at the "programming-in-the-small" level (e.g. at the level of individual procedures). If this were proven, then a separate study could examine how failure diversity "in the small" is related, if at all, to failure diversity at the system level.

6 Discussion

We have shown that there are traps in the intuitive ways of reasoning about the effectiveness of diversity-seeking decisions (DSDs) in project management, and there is room for progress towards more scientific decision-making. Our first contribution is simply to point out the possible pitfalls in intuitive reasoning: this knowledge in itself is a safeguard for the decision maker.

We have explored the intriguing possibility of giving a scientific basis to the common belief that - at least in some cases - failure diversity can be expected as a result of "fault diversity". If this could be shown for specific DSDs, decisions could be based to some extent on general laws of human error behaviour instead of having to be re-validated for every combination of processes and product requirements.

There is no guarantee of success, but we are working on specifying experiments to provide more empirical evidence. Initial experiments and analysis of existing data should aim at excluding un-promising theories and concentrate effort on those that may give practically useful rules for project decisions.

As for the example models in Section 5, the first two scenarios are unlikely, but we conjecture that the main theorems we used would still be true under less restrictive assumptions: we are working on mathematically tractable models for more general and realistic classes of situations. For instance, the formulas in Section 5 assume that all failure regions are fully disjoint. Yet, it is clear that similar formulas should apply if failure regions are *usually* disjoint, and a way of defining this "usually" should be adopted that can be verified by practical statistical observations. The third scenario (in 5.3) is much more plausible, though making it a practical possibility would still require much new experimental work.

The methods we have outlined for evaluating DSDs will be appropriate in those cases in which tractable models can represent the essential terms of the situation (all the important factors determining the probability of common failures for the specific system of interest, though not all factors), and parameters can be estimated with enough precision (the actual bounds on these estimates determine whether the method will show that the DSD is useful). There will certainly remain useful DSDs for which it will be impossible to prove their usefulness with the methods discussed here. Whether these will still be worth adopting will depend on the other evidence available and their costs.

This discussion has many limits. It is a preliminary discussion, about questions that have not been asked before in such detailed terms. For reasons of space, we have omitted considerations of several additional aspects of the problem which we believe to be ripe for investigation. For instance:

- we only talked about the average reliability to be expected from a certain DSD. In reality, it would be desirable to have an idea of the distributions of these probabilities - an idea of how likely we are to satisfy a given reliability requirement. We have shown elsewhere [20] that diversity may be seen essentially as a means for reducing the unavoidable *variance* in the results of software development. We are currently extending this line of research. However, even indications about averages would be more help for the decision makers than what is now available;
- there are ways a DSD may affect failure diversity that we have not yet considered. For instance, differences in structure between two complex versions may imply that essentially similar defects, even if triggered by similar sets of demands, produce errors that propagate differently in the two versions, resulting in failure diversity;
- we do not know yet how to reason about *combinations* of multiple DSDs. For some practical situations, it would be sufficient to be able to demonstrate that combining two given DSDs is no worse than applying either one of them alone. This seems a plausible conjecture, but we intend to clarify which conditions must apply for it to be true.

In any case, the necessary research involves the collection of empirical evidence, guided by models determining which kinds of evidence are needed and what the statistics should be like in order to demonstrate usefulness (or lack thereof) of specific project decisions.

Acknowledgements

This work was supported in part by Scottish Nuclear under the DISPO project (DIVERSE SOFTWARE PROJECT, Contract No. PP/79405/MB), and by the U.K. Engineering and Physical Sciences Research Council under the DISCS project (DIVERSITY IN SAFETY CRITICAL SOFTWARE, grant GR/L07673).

References

1. Voges, U. (Ed.): Software diversity in computerized control systems. Springer-Verlag, Wien (1988)
2. Lyu, M.R. (Ed.): Software Fault Tolerance. Wiley (1995)
3. Littlewood, B., Miller, D.R.: Conceptual Modelling of Coincident Failures in Multi-Version Software. IEEE Transactions on Software Engineering SE-15 (1989) 1596-1614
4. Littlewood, B.: The impact of diversity upon common mode failures. Reliability Engineering and System Safety 51 (1996) 101-113
5. Popov, P., Strigini, L.: Conceptual models for the reliability of diverse systems - new results. In Proc. 28th International Symposium on Fault-Tolerant Computing (FTCS-28), Munich, Germany (1998) 80-89
6. Lyu, M.R., He, Y.: Improving the N-Version Programming Process Through the Evolution of a Design Paradigm. IEEE Transactions on Reliability R-42 (1993) 179-189
7. MoD 00-55 Def Stan 00-55, Requirements for Safety Related Software in Defence Equipment. U.K. Ministry of Defence, Issue 2 (1997)
8. MoD 00-56 Def Stan 00-56, Safety Management Requirements for Defence Systems. U.K. Ministry of Defence, Issue 2 (1996)
9. Littlewood, B., Popov, P., Strigini, L.: A note on reliability estimation of functionally diverse systems. Reliability Engineering and System Safety, to appear (1999)
10. Reason, J.: Human Error. Cambridge University Press (1990)
11. Lyu, M.R., Chen, J., Avizienis, A.: Experience in Metrics and Measurements for N-Version Programming. International Journal of Reliability, Quality and Safety Engineering 1 (1994) 41-62
12. Avizienis, A., Lyu, M.R., Schuetz, W.: In search of effective diversity: A six-language study of fault-tolerant flight control software. In Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, Japan (1988) 15-22
13. Kersken, M., Saglietti, F. (Ed.): Software Fault Tolerance: Achievement and Assessment Strategies. Springer-Verlag (1992)
14. Mongardi, G.: Dependable Computing for Railway Control Systems. In Proc. 3rd IFIP Int. Working Conference on Dependable Computing for Critical Applications (DCCA-3), Mondello, Italy (1993) 255-277

15. Briere, D., Traverse, P.: Airbus A320/A330/A340 Electrical Flight Controls - A Family Of Fault-Tolerant Systems. In Proc. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, 22 - 24 (1993) 616-623
16. Kantz, H., Koza, C.: The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity. In Proc. 25th IEEE Annual International Symposium on Fault -Tolerant Computing (FTCS-25), Pasadena, California (1995) 453-458
17. Bishop, P.G., Pullen, F.D.: Failure Masking: A Source of Failure Dependency in Multi-version Programs. In Proc. 1st IFIP Int. Working Conference on Dependable Computing for Critical Applications (DCCA-1), Santa Barbara, USA (1989) 53-73
18. Chillarege, R.: Orthogonal Defect Classification. In Lyu, M.R. (Ed.): Handbook of Software Reliability Engineering: Computing, McGraw-Hill and IEEE Computer Society Press, (1996) 359-400
19. Bertolino, A., Strigini, L.: Assessing the risk due to software faults: estimates of failure rate vs evidence of perfection. *Software Testing, Verification and Reliability* 8 (1998)
20. Popov, P., Strigini, L., Pizza, M.: The efficacy of diverse redundancy against design error: some practical considerations. In Proc. INucE Third International Conference on Control and Instrumentation in Nuclear Installations, Edinburgh, U.K. (1998)