An Empirical Study of the Effectiveness of "Forcing" Diversity Based on a Large Population of Diverse Programs

Peter Popov, Vladimir Stankovic, Lorenzo Strigini
Centre for Software Reliability
City University London
London EC1V 0HB, U.K.
{ptp, v.stankovic, l.strigini}@csr.city.ac.uk

***Abstract*—Use of diverse software components is a viable defence against common-mode failures in redundant software-based systems. Various forms of "Diversity-Seeking Decisions" ("DSDs") can be applied to the process of developing, or procuring, redundant components, to improve the chances of the resulting components not failing on the same demands. An open question is how effective these decisions, and their combinations, are for achieving large enough reliability gains. Using a large population of software programs, we studied experimentally the effectiveness of specific "DSDs" (and their combinations) mandating differences between redundant components. Some of these combinations produced much better improvements in system probability of failure per demand (PFD) than "uncontrolled" diversity did. Yet, our findings suggest that the gains from such "DSDs" vary significantly between them and between the application problems studied. The relationship between DSDs and system PFD is complex and does not allow for simple universal rules (e.g. "the more diversity the better") to apply.**

Keywords-design diversity; multiple version software; software fault tolerance; diversity-seeking decisions; reliability improvement; experimental study

I. INTRODUCTION

Software diversity has been and is widely used in safety critical industries that require high software dependability and/or high assurance that a software dependability target has been met. While there is still controversy about the cost-effectiveness of diversity compared to other ways of improving dependability, in various industrial applications, diversity, rather than simple, non-diverse redundancy, is in practice a requirement for computer-implemented safety-critical functions.

Software diversity has been studied extensively, both theoretically and empirically; practical recommendations are available about how to apply it in industrial contexts [1, 2]. These include means for making the developments of diverse programs as 'independent' as possible; and ways of 'forcing' diversity between them by mandating differences between their designs (including "functional" diversity, in which the redundant and diverse subsystems satisfy a common system-level goal by implementing different specifications, e.g. in a nuclear protection system two functionally diverse subsystems typically use readings of different physical variables, say temperature and pressure, as inputs, and different algorithms, possibly based on different physical laws; see [3] for a definition) and their development processes. References to the relevant literature with many detailed recommendations are given in [4, 5]. We call "diversity seeking decisions" (DSDs) all these decisions that can be applied in the development or procurement of systems that are meant to be diverse. These recommendations are typically based on common sense or 'engineering judgment'; a critique based on extrapolations from theoretical work and from the scarce empirical data is given in [4]. Rarely, however, have these recommendations been backed by empirical evidence. It must be noted that DSDs generally try to achieve diversity between the products used in redundant configurations, e.g. their code structure or algorithms used, an indirect means towards the real goal of diversity between their *failure processes* when in operation in a specific environment [4].

We agree that proposed DSDs are usually supported by reasonable arguments, and indeed one of us co-authored the report [4] which attempted to clarify the likely effects of individual DSDs and the types of errors against which they could be effective. On the other hand, claims about the levels of improvement achieved are not now supported by experimental evidence. This is especially unsatisfactory since it is common in safety critical software that the fact of having applied measures to achieve high reliability or safety is taken as *prima facie* evidence of having achieved them.

For diversity, attempts have been made to associate various DSDs with numerical scores that the proponent of a diverse system could claim towards reaching a target level of diversity (reduction of the risk of common failures) [6]. There is unfortunately little or no supporting evidence to justify such scoring systems for either individual DSDs or their combinations. These rules, even if based on thorough surveys of industrial practice, can only codify opinion. Some empirical evidence of dependability improvements actually achieved by specific ways of forcing diversity would help. In particular, we suspect that even DSDs that are recommended on very reasonable grounds might produce very different degrees of improvement in different instances of their application. Measuring empirically the actual improvements

achieved in some application, and in particular when two DSDs are applied together to 'force' diversity, is the focus of this paper.

Several experiments have assessed the efficacy of software diversity in specific contexts. Most (including those that studied reasonable large samples - tens of diverse program versions) only studied "unforced" diversity: the only "diversity seeking decisions" applied were means for keeping the developments of the diverse program versions separate and as far as possible independent. Some experiments have studied DSDs for "forcing diversity", (especially diversity of programming language and of specification language - cf. [4] for references) but usually on small samples, and none - to our knowledge - have reported on the effectiveness of combining "diversity forcing" DSDs, as commonly recommended.

An important practical issue is whether combining different DSDs, e.g. using both different programming languages and different algorithms to implement the system function, tends to make a system more reliable than if only one of the DSDs (in the example cited, either different languages but the same algorithm or the same language but different algorithms) is applied, and how substantial the improvement, if any, is. Intuition suggests the principle "The more diversity the better", which has formed the basis for various practical recommendations [1, 7, 8]. Intuition, however, has repeatedly been wrong about design diversity. For instance, expecting independence between failures of software programs developed by independent teams was widely seen as plausible, but was convincingly demonstrated to be wrong – both theoretically [9, 10] and empirically [11].

The present work deals with the intuitive principle "The more diversity the better", and attempts to provide initial empirical evidence in favor or against it. It is known from theory [10, 12] that the principle is true in a limited sense: 'forcing' diversity, under specific probabilistic models with formally specified conditions of "everything else being equal", has been proved to be 'a good thing' - adding a further way in which the development processes of two programs are required to differ will make the expected PFD of the resulting 1-out-of-2[1] system better, or at worst leave it unchanged. But these "everything else being equal" conditions will only at times be realized, and only approximately, in real life, and there is a lack of concrete examples of actual degrees of improvement achieved.

To collect the initial evidence that we present, we took advantage of a large population of programs developed mostly by students. We do *not* propose our results as predictions of what will happen when applying a specific DSD in an industrial development for, say, safety critical applications. Rather, they are a starting point. First, this kind of empirical research shows patterns of behavior that may inspire new analyses and insight; secondly, they demonstrate, more strongly than theoretical considerations, that common-sense claims for DSDs may be misplaced. By showing that the *variability* of results that we think should be expected does occur in practice, they show that any claim for predictable effects of specific DSDs needs to be supported with appropriately strong and *specific arguments*.

Experiments in software engineering notoriously pose the dilemma whether to reproduce realistic industrial problems and development methods, with staff at industrial levels of competence, and incur high costs; or contain costs by using student manpower on small or even toy problems. The former choice leads to realistic case studies on *small samples*, so that there is *little confidence* that any effects observed are due to the factors studied rather than to some peculiarity in the sample; the latter allows satisfactorily large samples at the cost of possibly documenting patterns (e.g. mistakes and effectiveness of means for tolerating them) that are very different from those in an industrial project. In our study, we take advantage of the high number of programs in the sample, submitted by self-selected participants, and accept that the findings may not be directly applicable to an industrial context.

Yet, the effort seems worthwhile. Firstly, given widely held beliefs (usefulness of forcing diversity and of combining multiple DSDs) supported by little empirical evidence, a cheap experimental challenge to them is useful: if it rejects a commonly held belief (albeit only in a specific context), the resulting compelling indication that that belief needs more serious scrutiny is valuable and cheaply acquired. For instance, showing that sometimes forced diversity brings no advantage, or that it is not generally true that "the more diversity the better" (not just for some specific implementation of a system but statistically for a sizeable sample) would be such a valuable result. Likewise, as pointed out above, large observed variations in the observed effectiveness of some DSDs would forcefully refute the belief in constant levels of benefits, and demonstrate the need for specific evidence in each case. Results that confirm common beliefs would be admittedly weak confirmation, given the unrealistic settings, but obtained cheaply. Another useful output from cheap experiments is in the actual values of the measures of the benefits produced and how much they will vary: a few initial data points in an area where concrete measurements are lacking.

The rest of the paper is structured as follows: in section 2 we recall the theoretical and empirical work that underpins the research described in this paper. In section 3 we summarize the method used in our study. This is followed by the results about effectiveness of two single DSDs and their combination in Section 4. In the last section, we discuss the main findings of the research and the limits of our study, and outline possibilities for future work.

II. RELATED WORK

Design diversity has been studied very extensively - both theoretically and empirically.

Extensive experiments about diversity were funded, e.g. by NASA in the 1980s, [11, 13] and by the nuclear industry ([14, 15]) but few addressed alternative DSDs (ways of achieving diversity) and only on small samples. The experiments described in [11] demonstrated that failure independence between diverse programs cannot be assumed

[1] A system made up of two components, which only fails if both components fail.

in general, i.e. produced a counter-example for this theory, which had been proposed by some. This was, however, only a "single data point", and the validity of these experiments towards assessing DSDs in current development methods is limited. Cai et al. [16] have observed that present-day students given the specs from NASA experiments as exercises performed better than the programmers in the 1980s experiments, providing supportive evidence for design diversity, and suggesting that much has changed in programming culture (or that confounding factors abound in software engineering experiments).

Eckhardt and Lee developed an influential model (EL model) [9], which clarified why failure independence between diverse software cannot be assumed *a priori*. The model was generalized by Littlewood and Miller (LM model) [10] to take account of *forced diversity*, where a design authority requires different development teams to use different 'methodologies': for example different languages, testing or analysis techniques, etc. These models provided an important insight about use of diversity for fault tolerance: they explained that failure independence between diverse programs is just a possibility (probably unlikely), and that different levels of correlation (positive or negative) between failure behavior of diverse software can be observed in practice, including the possibility that forced diversity (LM model) leads to a system that performs better than if the constituent programs failed independently. More recent research, in part by the current authors and colleagues, has extended these insights (www.csr.city.ac.uk/diversity). This theory provides the basis for the work presented in this paper.

Experimental evidence of the effectiveness of DSDs meant to *force* diversity among different programs, and in particular combinations of such DSDs, is rather sparse. Meine van der Meulen initiated a range of empirical studies ([17, 18] and references therein) and also reported some initial results about the effect of a single DSD, the diversification of programming language [19]. We have re-used and extended his test harness and data analysis tools, and initially repeated his basic methodology, with some extensions.

III. METHOD

We studied a large pool of programs developed to various application specifications. To simulate how a system developer would try to "force" diversity among programs to be combined in a diverse-redundant system, we subdivided programs written to the same specification into sub-pools that differed according to two criteria, representing two possible DSDs that the system developer could apply. We then measured how often programs from different sub-pools failed together during extensive testing.

A. The UVa Online Judge Programming Platform

For the analyses described in this paper we used computer programs submitted to the *UVa Online Judge* platform (http://uva.onlinejudge.org/), maintained by Prof Miguel Revilla from the University of Valladolid in Spain and his team of collaborators worldwide. Any member of the public can use UVa Online Judge to submit *programs* that solve various types of computing "problems" in areas such as: sorting, number theory, graph traversal, etc. [20]. This repository includes programs submitted in 4 different programming languages: C, C++, Java and Pascal. As of May 2012 about 9.5 million programs have been submitted: C++ (64%), C (25%), Java (7%) and Pascal (4%).

B. Specifications ("Problems") Used in the Study

We have conducted the analysis for the following "problems"[2]:

- *Factors and Factorials*: A program written to this specification takes as input an integer N, 2<=N<=100, and must output the list of the number of times each prime number occurs in the factors of N's factorial. In total 13,526 programs were submitted, which we tested exhaustively on all 99 inputs;
- *3n+1*: A program written to this specification takes as input two integers {***i***, ***j***} and must calculate the maximum cycle length for each number between ***i*** and ***j*** (inclusively), where the cycle length is computed using the following algorithm:

```
... if n = 1 then STOP
       if n is odd then n = 3n+1
       else n = n/2 ...
```

  This is among the most popular "problems" from the UVa Online Judge platform. We tested more than 169,000 submitted programs on a test suite of 5000 tests with ***i*** chosen contiguously from the range [1, 100] and ***j*** from the range [1, 50].

All the "problems" studied specify programs that have no retained state between invocations.

We also analyzed a third "problem" - *Factovisors* (given in [21]). We do not include the results here for reasons of space and because they do not suggest different conclusions from the ones obtained for the *Factors and Factorials* and *3n+1* "problems".

For each of the "problems", our analysis included the following steps: i) compiling the source code of each program; ii) running the program executables on the test harness, on a suite of test cases (or "demands"), the same for all programs; iii) establishing and recording the *score* (success or failure) of each program on each demand; iv) performing statistical analysis of the raw scores.

We excluded from the analysis the programs that i) compiled to an executable binary that crashed upon starting, or ii) timed-out on 30 or more consecutive inputs. Among the programs submitted for the same "problem" by one author we selected *the first program that is not completely incorrect* (i.e. the program for which the output for at least one of demands is correct). All these aspects of the method are the same as in the earlier study [17]. As a consequence of the program selection method the pools of programs

[2] Details about each "problem" can be found at http://www.uvatoolkit.com/problemssolve.php

decreased to 4,290 for *Factors and Factorials*, and 15,463 for the *3n+1* "problem".

In the analysis of the results, the programs have been grouped according to different criteria:

- *Score classes*. Two programs belong to the same score class if they have identical scores (either correct output or failure) on every demand of the chosen test suite. We thus disregarded the differences between wrong outputs.
- *Sub-pool (SP)*. A subset of all programs submitted as solutions to a specific "problem" (or the 1-out-of-2 pairs built out of them), after applying a specific grouping criterion, e.g. a particular DSD or a combination of DSDs. For example, if we consider programming languages, separate sub-pools will be formed with programs written in Pascal, C++, etc.

C. DSDs Used in the Study

We have studied the following two categories of DSDs:

- DSD1 - forcing diversity by using different programming languages. This led to forming 4 sub-pools of programs: programs written in C, C++, Java and Pascal.
- DSD2 - forcing diversity of program structure. The purpose is to study the potential consequences of realistic DSDs such as mandating the use of different algorithms, which would plausibly lead to different program structures. We used a proxy measure of program structure based on two well known software 'complexity metrics': *Halstead Volume* (HV) [22] and *Cyclomatic Complexity* (CC) [23].

  Since these two measures are known to be highly positively correlated (and correlated with the simpler measure of size of code), we conjectured that the four sets of programs for which both are high, or both low, or that "violate" this positive correlation (low values of the one metric and high values of the other), may differ substantially in internal structure and thus possibly in algorithms used. This allows a quick automatic classification of the programs, though of course any conclusions based on this classification are only suggestive and to be confirmed by analysis of the actual algorithms in the programs.

  Thus, after computing the values of HV and CC for each program, we divided the population of programs (or the sub-pools resulting from DSD1) into four new sub-pools characterized by whether their values of the HV and CC metrics are above or below the medians of the two observed distributions of these two metrics. So, four mutually exclusive sub-pools were created: i) $SP_{HighHV,HighCC}$ - programs with **high HV, high CC** values; ii) $SP_{HighHV,LowCC}$ - programs with **high HV, low CC** values; iii) $SP_{LowHV,HighCC}$ - programs with **low HV, high CC** values and iv) $SP_{LowHV,LowCC}$ - programs with **low HV, low CC values**.

D. Pools of Programs

For every "problem" investigated, we initially analysed the effectiveness of design diversity by creating 1-out-of-2 pairs *without* forcing diversity (i.e., the only DSD applied was that the programs were developed by different authors), by selecting the pairs randomly from the *population of all programs*, as modelled by the EL model [9].

In contrast, to assess "forced" diversity, the programs used to form a 1-out-of-2 system are drawn from two different sub-pools of programs, which model the impact of one diversity-forcing DSD, or of a combination of two. For instance, in case DSD1 is used, say with programming languages C++ and Pascal, we would first form the sub-pools of C++ and of Pascal programs, and then form the pairs by selecting at random one of the programs from the C++ sub-pool and the other from the Pascal sub-pool. This situation is modelled by the LM model [10].

Further, if only DSD2 is used, the sub-pools would be derived from the pool of all available programs. The pairs of programs are then formed by selecting programs from different sub-pools, e.g. from $SP_{HighHV,HighCC}$ and $SP_{HighHV,LowCC}$, respectively.

Finally, to combine the two DSDs, we formed the sub-pools by applying the two DSDs in order, i.e. *first* applying DSD1 as described above, producing sub-pools for the four programming languages; *then* applying DSD2 to each of these, dividing the programs written in a given programming language into 4 sub-pools according to the values of HV and CC. With 4 programming language pools, this created the following 16 sub-pools: SP1 - programs written in **C**, with **high HV**, **high CC** values; SP2 - programs written in **C**, with **high HV**, **low CC** values; SP3 - programs written in **C**, with **low HV, high CC** values; etc.

E. Measure of Software Reliability Improvement through Diversity

Irrespective of whether we were considering sub-pools or the whole set of programs, we simulated the effects of *reliability improvement* during development of a system, or of development processes (that produce the program versions) of varying - increasing - quality. This improvement is simulated by considering progressively fewer programs from the least reliable score class in that pool or sub-pool. In other words, considering the statistics measured on the programs that were actually submitted simulates the results of a development process in which the probabilities of producing programs belonging to each score class are the same as their observed frequencies; we simulate improvement by reducing the probability of the programs in the least reliable score class; when that class becomes empty, we move on to the next least reliable score class. Further details of the procedure can be found in [21]. When drawing programs from diverse sub-pools (to examine the effectiveness of *forced* diversity), we ensured that the two sub-pools had been reduced to the same average single-program PFD.

There are alternative ways of simulating reliability improvement, or variations in the quality of the development process. In the most general approach, one could:

(i) consider the pool of submitted programs as providing just a representative pool of possible versions with their faults, but (ii) assign them probabilities that match not their frequencies in the submitted pool but various conjectures about how likely the various kinds of programs (or of faults) would be in various plausible development processes. For instance, one could assume that even a process with very high average achieved reliability may still produce (with low probability) programs from the least reliable score class; or that a process improvement would consist of adding a verification procedure that practically eliminates a certain class of faults. For instance, considering the importance of static analysis in industrial practice, we could apply a static analysis procedure to the source code of the programs and discard those identified as problematic, even if they happen to belong to highly reliable score classes.

These various options are being considered for continuing the present study.

In this phase of the study we assess the *reliability improvement ratio* R:

$$R = PFD_A / PFD_{AB}, \quad (1)$$

between the average PFD value of the individual programs used to form 1-out-of-2 pairs (the whole population or specific sub-pools) and the average PFD of the pairs so obtained [17, 18]. While the real target measure for a system is its probability of failure per demand, and a development method could be assessed by the mean, PFD_{AB}, that it delivers, R seems a plausible indicative measure for ranking alternative means for reliability improvement.

IV. RESULTS

The sub-sections below summarize our observations. The plots apply the following visualization conventions: all plots have the same range of values for the scale of the Y axis, but we add two further levels to show i) very large values of R (i.e., values that exceed the largest number shown on the scale); ii) "infinite gain" (i.e., all program pairs exhibit no failures: the denominator of the reliability improvement ratio (PFD_A/PFD_{AB}) is zero). Each plot also shows a line for the hypothetical reliability improvement if the programs failed independently, i.e., if $PFD_{AB} = PFD_A * PFD_B$, as in many discussions we have found others to use this product as a reference value. However, we wish to emphasize that the measure of interest is how reliable a system is, not how it compares with a hypothetical model of how reliable it could be.

A. The Factors and Factorials "Problem"

Fig. 1 summarises the findings for the *Factors and Factorials* "problem". Fig. 1a – no forced diversity – shows how the ratio, R, changes with the change of the average PFD_A. Interestingly, when the pool of programs is less reliable (right-hand side of the plot), the reliability improvement is close to "failure independence"; when the pool becomes more reliable, the gain remains substantial and tends to increase, although with ups and downs.

Fig. 1b illustrates the effect of forcing diversity using DSD1 only. One program in the pair is a **Pascal** program, the second program is chosen at random from the sub-pools of C, C++, Java and Pascal programs. The curves representing different combinations of diverse languages are very close, i.e., the 3 instances of DSD1 do not substantially improve system reliability. All curves shown are very close to the *Pascal, Pascal* curve and to the curve shown in Fig. 1a where neither DSD1 nor DSD2 was applied[3].

Fig. 1c shows effects of forcing diversity by DSD2 only. The plot shows 4 curves in which the first program is chosen from the sub-pool with **low HV** values and **low CC** values, while the second program is chosen from one of the 4 sub-pools, respectively. As with Fig. 1b, forcing diversity does not lead to significant differences between the sub-pools of pairs. Intriguingly, the curves indicate that the pairs with no apparent structural diversity (both programs come from the pool **low HV**, **low CC**) turn out to be the most reliable, for average PFD values of about 0.0004 or less.

Finally, Fig. 1d shows the effect of combining DSD1 and DSD2. The most reliable pairs on average (represented by the curve with diamond symbols) are formed from **C** programs with **high HV** and **high CC** (the *reference sub-pool* in this plot) and **Java** programs with **high HV** and **high CC**. As reliability of single programs improves, system PFD gets even better than if the component programs failed independently. Interestingly, the 'structure' (complexity) metric values for the single programs (one implemented in C, the other in Java) forming these pairs come from the **high HV, high CC** sub-pools, that is, there is no evidence of diversity in structure between the two programs. Somehow, reducing the sub-pools of C and Java by filtering out the programs with low values of either HV or CC resulted in much more reliable pairs than if the non-filtered sub-pools of C and Java were used. That is, when we combine programs written in C and in Java, we find that in this set those with higher values of HV and CC are 'more diverse' than those with low values of either of the two software metrics. A conjecture would be just that writing longer programs gives programmers room for a greater variety of mistakes.

Another set of diverse program pairs (represented by the curve with asterisk symbols) is formed by choosing the first program from the reference sub-pool, and the second one from Java programs that have low HV and low CC. In this case, combining two DSDs brings significant benefits – when the average PFD of the single programs is better than 10^{-3}, average system reliability is 2-3 orders of magnitude better, still a very significant improvement despite it not being the highest observed gain.

Readers may notice that many curves converge to the value 100 at the left end of the X axis, where the average PFD of single programs is 10^{-4}, and wonder whether this is an artefact of the data collection method. It is actually a consequence of the set of faults present in these programs, and the way the experiment simulates how average

[3] Although they are based on different populations: Fig. 1a shows the effects of selecting from programs written in all four languages; Fig. 1b from two of them at a time.

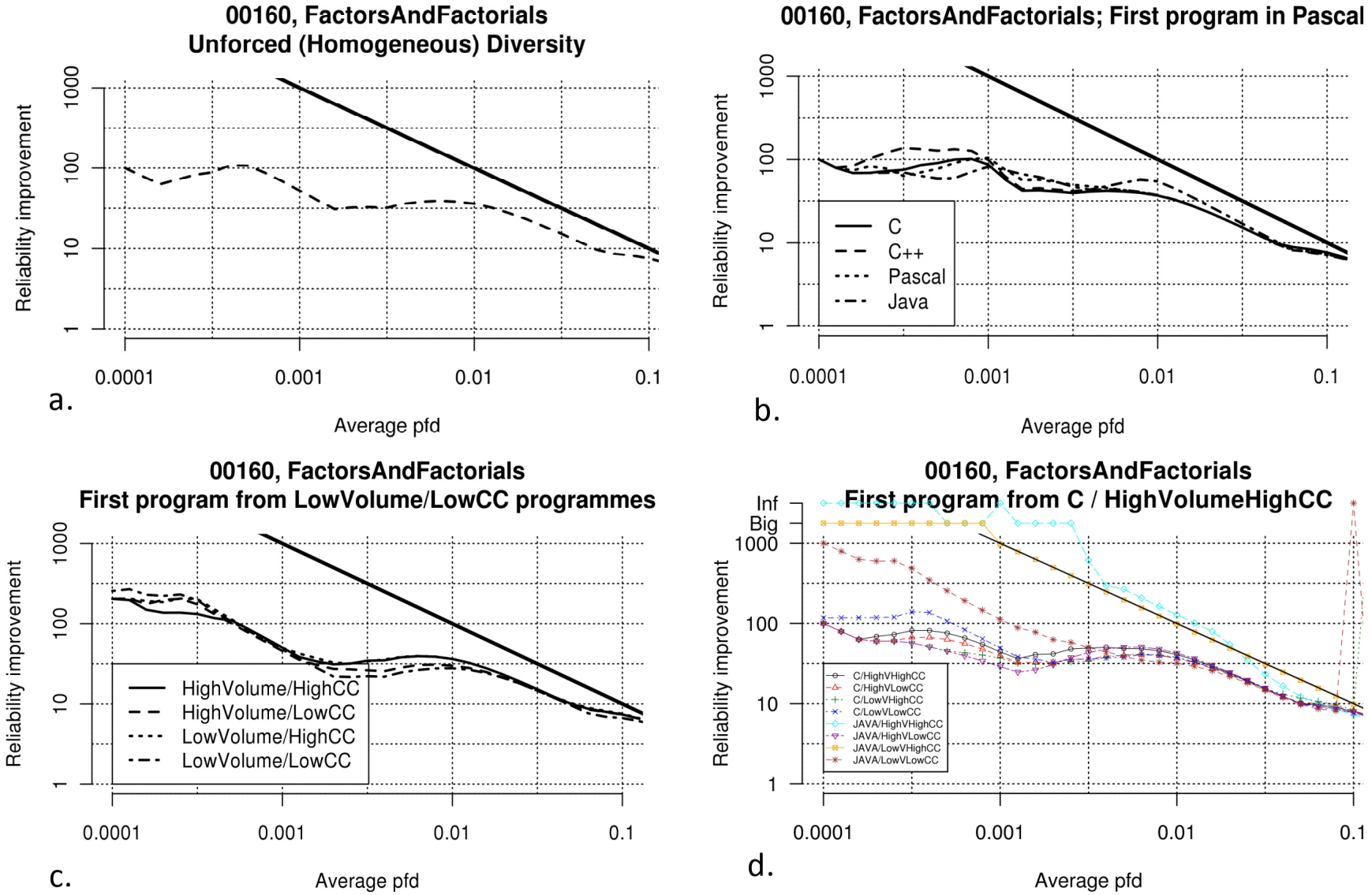

Figure 1. Examples of effects of diversity for the *Factors and Factorials* "problem", in the cases of: unforced diversity (a); one of the two DSD categories, either diverse programming language (b), or diverse program structure (c) is used individually; or when the instances of two DSD categories are combined (d). The horizontal axis shows either the average PFD of the whole pool of programs (a), or the average PFD of the two sub-pools from which the programs (A and B) are selected (b, c and d). The vertical axis shows the reliability improvement as the ratio PFD_A/PFD_{AB}.

program quality improves. If we rank the faults in these programs by their "size" (associated PFD value), we find that there is only one fault with the smallest measured non-zero PFD, say PFD_{min}. As we reduce the program pool to achieve lower average PFD, the pool is eventually reduced to containing just two score classes: programs with PFD=0, and programs with this single "smallest" fault. It can be shown that in these conditions the reliability gain would indeed converge to PFD_{min}/(average PFD of the pool).[4] In this set, $PFD_{min}=10^{-2}$ and thus the R value, for $x=10^{-4}$, is $10^{-2}/10^{-4}=100$.

B. The 3n+1 "Problem"

As in the previous section, Fig. 2a shows how the average population reliability affects the gain from software fault tolerance: here, too, the gain generally increases with improving reliability of single programs, although the increase is not monotonic throughout the range. For low reliability (the right-hand side of the plot for values of the average PFD in the range 0.01 to 0.1), the gain suggests "failure independence" between programs, but this changes with the improvement of reliability and the distance of the gain from "independence" is increasing. The gain peaks at about 236 for average PFD of 0.0002 – the highest gain observed for any of the "problems" when not forcing diversity.

Fig. 2b illustrates the effect of applying only DSD1 as the sub-pools' reliabilities change. The reference sub-pool this time contains programs written in **C**. For most of the reliability range shown, the best performing combination is given by pairs from the reference sub-pool and the sub-pool of **Pascal** programs (down to values of PFD of the sub-pools of about 0.0003). For lower values of the average sub-pool PFD the **Java** sub-pool "replaces" the sub-pool of Pascal programs. The differences between average systems PFDs are at times large – they reach one order of magnitude when the average PFD of the pools is in the range [0.01, 0.0016].

Fig. 2c shows the effectiveness of using solely DSD2. It is interesting that when the average PFD is 0.001 or lower, the highest reliability gain is observed when we pair programs from the reference sub-pool (**high HV**, **high CC**) with the diverse sub-pool in which the programs have "low" values for both software metrics.

Fig. 2d shows the effect of combining instances of two DSDs. The reference sub-pool is **C, high HV, high CC**. The best combination is obtained when the second program comes from the sub-pool **Pascal, low HV, low CC**. Thus, in this case, "the more diversity the better". Interestingly, for a range of PFD values, the gain is better than under "independence on average". The second-best combination (almost consistently for all values of average PFD on the X axis) is formed with a second program coming from the sub-pool **Pascal, high HV, high CC**. This is interesting because it shows that the difference in structure does not necessarily contribute to further gain. Pascal programs with high HV, high CC turn out to be a "better fit" for C, high HV, high CC than Pascal, high HV, low CC, or Pascal, low HV, high CC. The fact that two of the Pascal sub-pools give good failure diversity with the reference sub-pool is not surprising – Fig. 2b already suggested that the Pascal programs are a good fit to the C programs from this viewpoint. Finally, the role of structure is emphasised by the fact that the system formed when the second program comes from the sub-pool **C, low HV, low CC** achieves high values of the gain: despite the use of the same language, the structure of the C programs *may* be strongly related to which faults they are likely to contain and thus significantly affect reliability of 1-out-of-2 systems.

[4] For two pools of *different* programs, and if the programs with the smallest non-zero PFD in each pool are such that their failure sets overlap, PFD_{min} should be read as the PFD of a pair made from one instance of such programs from each one of the two pools.

V. Conclusions and Future Work

This paper presents initial empirical findings about the efficacy of some "diversity seeking decisions", obtained using a large number of software solutions to mathematical "problems" from the UVa Online Judge platform. We performed exploratory analysis "on average", in which we studied empirically how two different individual DSDs and combinations thereof affected the reliability of 1-out-of-2 system. We report on:

- analysis of the effectiveness of an additional DSD, not studied before – diverse software program structure. We used a combination of software complexity metrics as a proxy for diverse program structure.
- analysis of the effectiveness of a combination of two "diversity-forcing" DSDs: diverse programming language and diverse program structure.

This study, the first to systematically analyse the effects of combining different ways of 'forcing' diversity, has produced some useful results by showing *high variability* of benefits from each of the two DSDs studied and from their combination. It gives a *proof of existence* of cases in which "more diversity" is beneficial, i.e. these findings provide empirical evidence that using particular instances of the two categories of DSDs may be more effective than using an instance of a single DSD, or of not forcing diversity at all. To the best of our knowledge, this is the first reported empirical evidence of this kind. There are, however, counter-examples: the results showed many cases where applying instances of both DSDs together led to worse systems than with just one DSD. Together, all this is evidence of significant variation in the effectiveness of combined DSDs: something that does not contradict any previous knowledge but had not been shown previously.

How effective a set of DSDs should be expected to be for a particular system depends in the end on the types of faults that the specific development processes are most likely to leave in the diverse software components. Therefore, one should not just assume *a priori* that applying a specific combination of DSDs will result in a specific level of system reliability gains or achieved system reliability; at the system assessment stage (e.g. for certification) one should be prepared to provide direct evidence of the dependability achieved in the specific system.

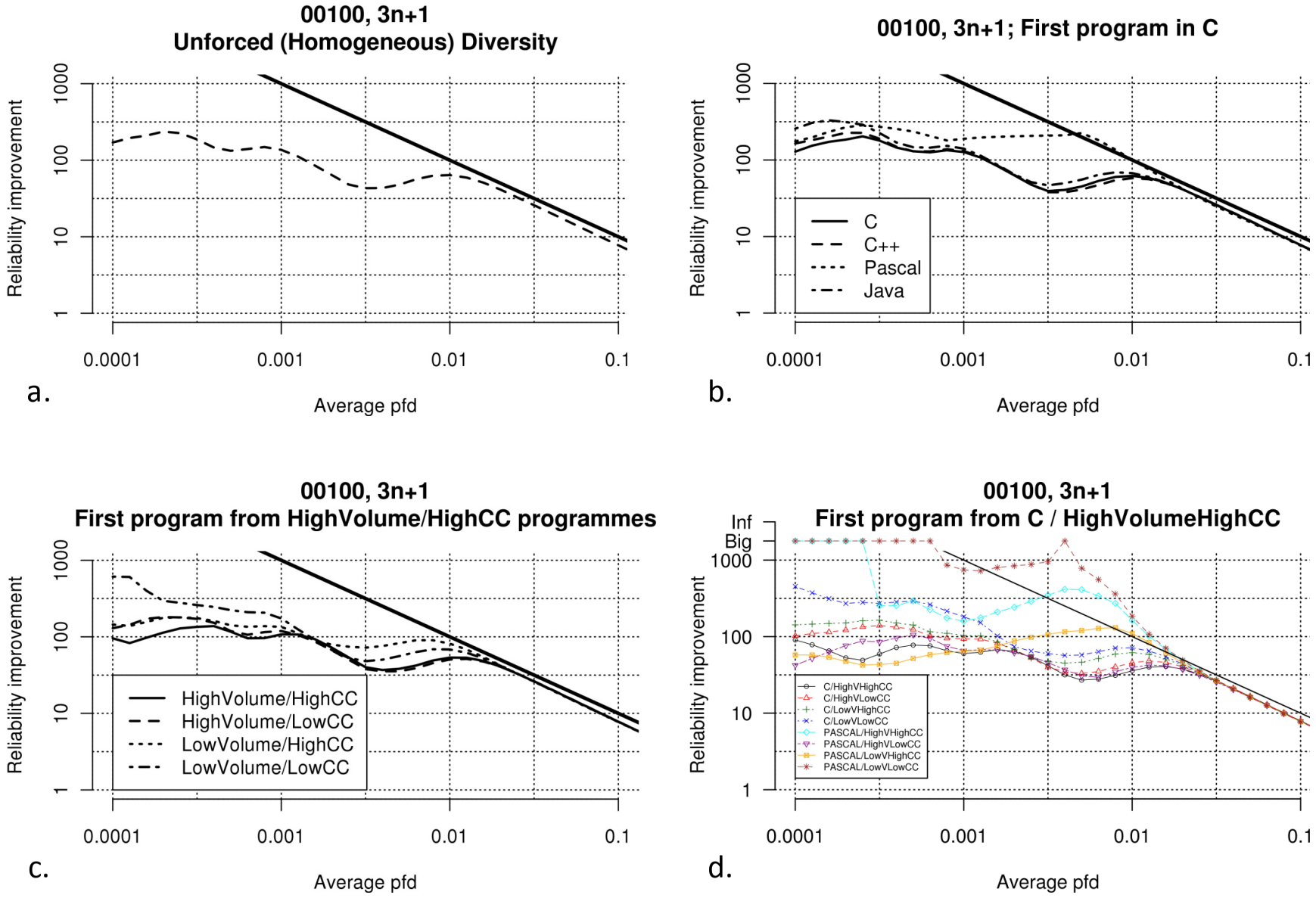

Figure 2. Examples of effects of diversity for the 3n+1 "problem", for the four different cases described in Fig. 1. The plot shows average reliability improvement of a diverse pair, relative to an individual program (A), when: both programs are from the whole pool (a); the "first" program (A) is from the sub-pool of C programs (b); the "first" program (A) is from the sub-pool of programs with high HV and high CC values (c), or the "first" program (A) is from the programs belonging to SP1 (d). The meanings of the axes are given in Fig. 1.

The limits of this study belong to two categories: limits of the dataset and limits of the method. Regarding the data set: we deal with solutions developed by *self-selected programmers*, unlikely to have followed a rigorous process. Many of the submitted program versions are of very low reliability. There is no *a priori* reason to expect that specific measurements obtained with these (simple) programs would extend to others, and in particular to ones with high reliability requirements, so that they could inform decisions about production software, which is frequently more complex, and/or developed much more rigorously, and/or with more advance thought given to how to seek diversity. In industrial practice, diversity is sometimes achieved by using radically different programming styles or specifications, e.g. ELEKTRA [24] uses two channels with conventional and rule-based programming. On the other hand, the degree of language diversity represented here is akin to that used in certain industrial developments. Extending direct empirical investigation to other aspects of programming language diversity (e.g., imperative vs PLC-oriented graphical languages) would certainly be beneficial if appropriate data sets can be obtained. Our results indicate heavy variability of the reliability gain obtained via diversity-forcing DSDs. A combination of DSDs we explored in some cases gives much greater gains than that achieved by not forcing diversity (the latter is shown in Fig. 1a and Fig. 2a); the results of combining DSDs provide suggestive evidence of significant variation in their effectiveness (there exist examples of combined DSDs being less effective than the instances of a single DSD). Although this is only an initial data point, it indicates that any expectation of consistent reliability gains from a method for forcing diversity would need to be supported by further investigation.

Some limits of the method of our study, such as the proxy used for diverse structure, are addressed below.

The main ways in which we consider extending this study are:

- Analysing more "problems" within the *UVa Online Judge* platform, to explore further the range of effects (such results are useful for practice to avoid unwarranted assumptions) and to shed light on whether common "empirical laws" apply to ranges of applications, program complexity and/or program quality, i.e. whether there is a range of effects of DSDs that apply across multiple "problems", or instead the gains are significantly dependent on the specific "problem". Extensive data from such analyses will allow us to quantify more extensively the range of variation of the effectiveness of combining multiple DSDs.
- Extending the testing by more extensive coverage of the demand space. Programs for *3n+1* and *Factovisors* "problems" were tested exhaustively over a *subset* of their demand spaces, so that faults affecting this subset are explored in complete detail. A complementary picture would be obtained by:
    - Exhaustive testing of the whole demand space (this was done for the *Factors and Factorials* "problem"), in cases where this is feasible.
    - Sampling from the whole demand space according to some distribution. For production software systems, this distribution is meant to be representative of the operational profile, i.e. a quantitative characterization of how a system will be used. When this is not known, uniform sampling, or alternative, hypothetical but plausible profiles of use, would give evidence of effects (or lack thereof) of the profile on the findings.
- Pre-selecting programs that have passed more extensive acceptance tests, so that the data collected concern only the programs of higher reliability within the pool, akin to the reliability levels that are likely in industrial practice.
- Simulating alternative ways of varying the quality of the software development process (as explained in Section III.E) by changing the frequencies of each individual score class (to be different from the frequencies obtained from the experimental data).
- Using more refined techniques to divide the programs into sub-pools in meaningful ways. A clear candidate is how to detect likely differences in algorithms or in organisation of programs. We could stick to a 'blind' software metric based approach, but applying more refined techniques for clustering multi-dimensional data sets. Alternatively, we could automatically identify actual differences in program structure, using, for example, reverse engineering tools, or plagiarism detection tools used in teaching programming.

ACKNOWLEDGMENT

This work was supported in part by the DISPO project, funded under the CINIF Nuclear Research Programme by EDF Energy Limited, Nuclear Decommissioning Authority (Sellafield Ltd, Magnox Ltd), AWE plc and Urenco UK Ltd. ("the Parties"). The views expressed in this paper are those of the author(s) and do not necessarily represent the views of the members of the Parties. The Parties do not accept liability for any damage or loss incurred as a result of the information contained in this paper.

We are grateful to Meine van der Meulen for providing the full data and the code from his PhD research, including the analysis tools we have re-used and extended; and to Miguel Revilla for making the research possible in the first place by creating the UVa Online Judge as an open tool for lifelong learning in computer science, and then granting us access to the pool of programs.

REFERENCES

[1] Preckshot, G.G., "Method for Performing Diversity and Defense-in-Depth Analyses of Reactor Protection Systems", NUREG CR-6303, U.S. Nuclear Regulatory Commission, 1994.

[2] Wood, R.T., R. Belles, M.S. Cetiner, D.E. Holcomb, K. Korsah, A.S. Loebl, et al., "Diversity Strategies for Nuclear Power Plant Instrumentation and Control Systems", NUREG/CR-7007, U.S. Nuclear Regulatory Commission, Washington DC, 2010.

[3] IEC, "Instrumentation and Control Systems Important to Safety-Requirements to Cope with Common Cause Failure (CCF)", IEC 62340, Geneva, Switzerland, 2008.

[4] Littlewood, B. and L. Strigini, "A Discussion of Practices for Enhancing Diversity in Software Designs", DISPO project reports, LS_DI_TR-04, City University, London, 2000, http://www.csr.city.ac.uk/diversity/Papers/LS_DI_TR04/.

[5] Popov, P.T., L. Strigini and A.B. Romanovsky. "Choosing Effective Methods for Design Diversity - How to Progress from Intuition to Science". in the 18th International Conference on Computer Safety, Reliability and Security. Toulouse, France, p. 272-285, 1999.

[6] Waterman, M.E. and R.T. Wood. "Evaluating Diversity in Digital System Designs". in NPIC&HMIT 2009, Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies. Knoxville, Tennessee, 2009.

[7] Lyu, M.R. and Y. He, "Improving the N-Version Programming Process through the Evolution of a Design Paradigm". IEEE Transactions on Reliability, 42(2): p. 179-89. 1993.

[8] Saglietti, F., "A Classification of Software Diversity Degrees Induced by an Analysis of Fault Types to Be Tolerated", in the 5th International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems, Tests, Diagnosis, Fault Treatment. 1991, Springer-Verlag. p. 383-395.

[9] Eckhardt, D.E., Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors". IEEE Transactions on Software Engineering, SE-11(12): p. 1511-1517. 1985.

[10] Littlewood, B. and D.R. Miller, "Conceptual Modeling of Coincident Failures in Multiversion Software". IEEE Transactions on Software Engineering, 15(12): p. 1596-1614. 1989.

[11] Knight, J.C. and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming". IEEE Transactions on Software Engineering, 12(1): p. 96-109. 1986.

[12] Salako, K. and L. Strigini, "Diversity for Fault Tolerance: Effects of "Dependence" and Common Factors in Software Development", DISPO project reports, KS-DISPO5-01, Centre for Software Reliability, City University London, 2006.

[13] Eckhardt, D.E., A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability". IEEE Transactions on Software Engineering, 17(7): p. 692-702. 1991.

[14] Bishop, P.G., D.G. Esp, M. Barnes, P. Humphreys and G. Dahll, "PODS - a Project on Diverse Software". IEEE Transactions on Software Engineering, 12(9): p. 929-940. 1986.

[15] Lyu, M.R., Software Fault Tolerance. Trends in Software, ed. B. Krishnamurthy, John Wiley & Sons. 1995

[16] Cai, X., M.R. Lyu and M.A. Vouk. "An Experimental Evaluation on Reliability Features of N-Version Programming". in the 16th IEEE International Symposium on Software Reliability Engineering. Chicago, USA, p. 161-170, 2005.

[17] van der Meulen, M. "The Effectiveness of Software Diversity", PhD thesis, City University London, London. p. 226, 2007.

[18] van der Meulen, M.J.P. and M.A. Revilla, "The Effectiveness of Software Diversity in a Large Population of Programs". IEEE Transactions on Software Engineering, 34(6): p. 753-764. 2008.

[19] van der Meulen, M.J.P. and M. Revilla. "The Effectiveness of Choice of Programming Language as a Diversity Seeking Decision". in the 5th European Dependable Computing Conference (EDDC-5). Budapest, Hungary, p. 199-209, 2005.

[20] Revilla, M. and S. Skiena, Programming Challenges. 1 ed, Springer. p 368. 2003

[21] Popov, P., V. Stankovic and L. Strigini, "An Exploratory Study About the Effectiveness of "Diversity-Seeking Decisions" Based on a Large Population of Diverse Programs", DISPO project reports, City University London, London, 2012.

[22] Halstead, M.H., Elements of Software Science. Operating and Programming Systems. New York, NY, Elsevier Science Inc. 1977

[23] McCabe, T.J., "A Complexity Measure". IEEE Transactions on Software Engineering, SE-2(4): p. 308-320. 1976.

[24] Kantz, H. and C. Koza. "The Elektra Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity". in the 25th International Symposium on Fault-Tolerant Computing. Pasadena, California, p. 453-458, 1995.