Achieving effective diversity between redundant software-based components

Lorenzo Strigini

Centre for Software Reliability, City University, London

E-mail: L.Strigini@csr.city.ac.uk

Abstract

All empirical evidence indicates that diversity between redundant software-based components offers some defence against common-mode failure in redundant systems, i.e., it brings gains in reliability or safety. An important question is how to pursue diversity - in the selection or development of such software - so as to achieve large enough gains. Common sense suggests for instance to develop the components, or procure components that have been developed, in "truly independent" ways; and to make them "as diverse as possible", i.e. with intentional differences in their designs and development methods. This advice is unfortunately insufficient for most practical decisions, and turns out sometimes to be self-contradictory, while direct experimental evidence of "what really works" in industrial practice is scarce. This talk will summarise the state of knowledge on these issues:

- the ways diversity can be pursued, using a threat-driven approach to analysing the possible "diversity seeking decisions";
- the trade-offs that may arise given the practical constraints in an actual project, having to choose in a limited range of options; and especially the common case when the pursuit of diversity may work against that of high reliability of the individual channels, while the combined effect of these two factors on system-level reliability or safety, the true goal pursued, is difficult to estimate;
- mathematical results that in some cases are sufficient for choosing between alternative policies, even without specific experimental evidence. These are based on probabilistic models and identify scenarios under which pursuing some additional degree of either "separation" or "diversification" between the development processes of redundant components is guaranteed to yield improvements at the system level.

Presented at the 6th International Conference on Control and Instrumentation in Nuclear Installations, Manchester, 11th - 13th September 2007

1. Introduction

In redundant systems for critical applications, design faults, if repeated in all redundant computation channels, could cause common-mode failures. Diversity between the redundant channels provides some protection against this danger. Two current trends are increasing the importance of fault tolerance via diversity in all applications of computers: the push towards entrusting more critical functions to software-based components (due in part to their desirable features, and in part to the practical disappearance of non-software based alternatives); and the increased reliance on off-the-shelf products, which may lack sufficient evidence of the required reliability.

Diversity has mostly been studied for software, so we will refer to scenarios of software development, although the same principles apply to hardware diversity. Software diversity is sought by having two or more separately developed variants (often called *versions*, originating the term *N-version programming* for this use of diversity) of a program. The versions must exhibit the same functional (externally visible) behaviour. It is hoped that, if one version fails, the other, diverse version[s] will not fail at the same time; that is, it is hoped that any bugs they may contain will not cause failures in exactly the same circumstances in all versions. The two or more versions are run in a redundant configuration, so that failures of a subset of the versions may be masked or at least detected.

Diversity poses the same problems as all other techniques for defending against design faults to achieve high dependability: first, how to forecast their effectiveness; and then, since this forecasting is problematic, how to direct their application to make them effective and cost-effective. In this paper, we focus on the question of how best to pursue effective diversity, i.e., how to differentiate redundant channels to achieve a low probability that they will fail together.

In the discussion that follows we refer to the simplest scenario of redundancy with diversity: a parallel-redundant, 2-channel protection system, which, in a stylised representation, consists of two separate channels, each receiving sensor readings from the plant and able autonomously to cause a shut-down through a single, Boolean output. We are interested in the system's behaviour on demand (so, we do not consider spurious trips), and the measure of interest is the system's probability of failure on demand (*pfd*). This system is safe against any failure that affects only one of the diverse channels; while any failure of both channels is an unsafe failure of the system.

If this is a bespoke system, each version is produced by its own "version development team"; a "project manager" defines the requirement specifications that the development teams must implement and the constraints under which they have to work, handles specification updates and decides on final acceptance of the developed versions.

The reason for combining components in a redundant fashion is the potential "diversity of failures" between them: we are interested in how unlikely they are to fail together on the same demand. The goal is low correlation between their failures; "diversity" is an informal term for this goal. The word "diversity" is also used (somewhat confusingly) to indicate the factors that tend to reduce failure correlation, and especially those factors that developers can control. In earlier articles [6], we defined the special term "diversity seeking decision" ("DSD") to avoid the common confusion between *decisions* made about these factors and the (desired or actual) effects of these decisions, i.e., the degree of actual diversity between versions or between their failure behaviours. A DSD is a *decision* available to system designers or project managers to attempt to promote *failure diversity* between program versions. Recommendations about DSDs [1] have been made since the beginning of research in software design diversity. All DSDs belong roughly to two categories, with two different goals:

- to keep the development processes for the versions as separate and (informally speaking) "independent" as possible (preventing possible mistakes from "leaking" from one to the other[s]);
- and/or to make these development processes "more different" ("forcing" diversity). For instance, mandating different designs between two versions, and the use of different methods (including tools, languages, test processes, etc) in developing them.

These recommendations often refer to a scenario of bespoke development of (all versions of) a new system: the project manager can, in principle, decide all details of the development process. In reality, this freedom is always limited by practicality (e.g., skills of the available staff) and cost factors, and the need to rely on off-the-shelf components where possible. As a limiting case, a project may be limited to selecting completely off-the-shelf implementations of all the diverse channels. The DSDs are then applied to this selection process: e.g., there is usually a preference for components that have been developed independently by separate companies.

But how can a project manager choose from the long list of possible DSDs? It is not necessarily the case that the more DSDs are applied, the better, and there is a need to be selective, because DSDs may not be mutually compatible, and most have costs: duplication of activities, added co-ordination effort, need for staff with specific skills. Choosing, however, involves difficulties:

- the link between any DSD and the potential resulting reduction in the probability of common failures is unclear: how do we know which DSDs will be cost-effective?
- although it is natural to focus on maximising diversity between the channels, this may not maximise system reliability: decisions may

involve subtle trade-offs between diversity between the channels and dependability of the individual channels;

- even if we are satisfied with our estimate of the effect of an individual DSD, we actually have a choice between alternative *combinations* of DSD: a more complex decision problem

Guidance about these issues may be sought from several sources. Controlled experiments have been run, but their evidence is too sparse to support any general law about the effects of specific DSDs on failure diversity; their main utility has been in providing counter-examples to *refute* conjectured general laws[1]. General experience in software development does give experts some idea about how different software development methods are prone to different kinds of mistakes, although this knowledge does not include measures of these differences, and it concerns differences caused in patterns of *faults* rather than of *failures*. But for the more complex questions, there is no consensus, and intuition has repeatedly been shown to be misleading in this area. Help can then be sought from probabilistic models that give insight about what we *should* expect. These models lead to theorems of the form "*If* two alternative development processes for a multiple-version system satisfy a certain set of detailed conditions, and differ in some specific aspect of the combinations of DSDs they use, the first process is to be preferred to the second". Although these apply to restrictive set of conditions, and often help to identify a preference but not to assess the extent of the probable gains, they also help to clarify the unspoken assumptions implicit in informal judgements, and thus avoid possible fallacies. This paper draws on research performed in the DISPO (Diverse Software PrOject) projects over several years, including new results from probabilistic modelling.

In Section 2, we discuss how DSDs achieve their goal of promoting failure diversity, the limitations in available knowledge and a threat-based criterion for selecting DSDs; in Section 3, we discuss, with the aid of results from probabilistic modelling, the usefulness of separation and independence between version developments, the ways of "forcing" diversity even when these are not assured, and some preference criteria between alternative forms of diverse developments. Section 4 deals with the especially difficult issue of choosing *combinations* of DSDs. Section 5 contains some conclusions.

[1] Many controlled experiments were run in the 1980s (see [5] for references). As usual in software engineering, there are so many possible variations between development processes that generalising from their results is always suspect, except as proofs of existence or counter-examples. In these experiments, diversity has always produced reliability improvements, and it has been demonstrated that independence between development activities of different versions does not imply statistical independence of failures.

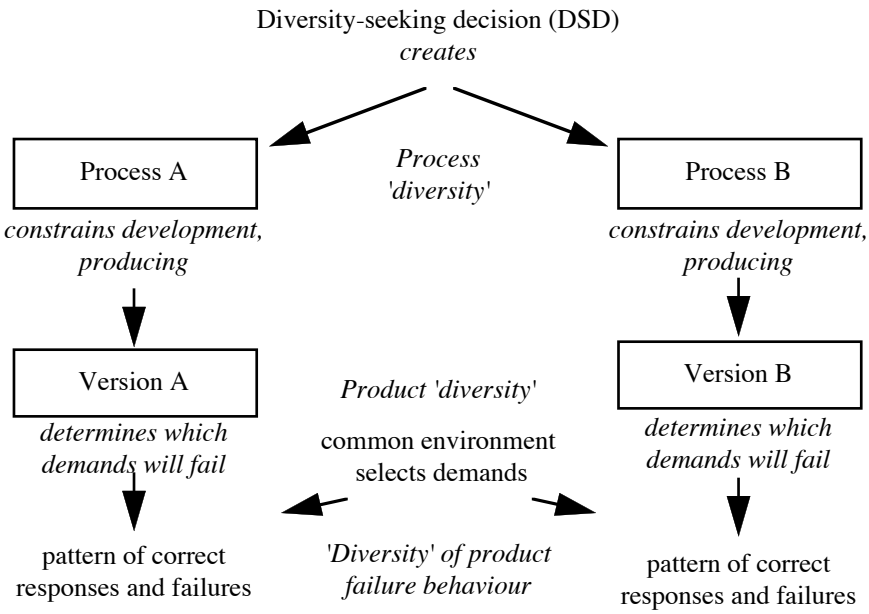

Fig. 1. From diversity-seeking decisions to failure diversity

2. Diversity seeking decisions

Purposes and effects of DSDs

In a DISPO project report [4], we discussed the various DSDs available to a project manager who wished to produce diverse versions of computing channels. Considering how effective a certain DSD would be in improving failure diversity and, in the end, system *pfd*, means considering the long cause-effect chain sketched in Fig. 1. This means reasoning about which factors affect the likelihood of various human errors in development, and how these errors affect software faults. Unfortunately there is little empirical, quantitative knowledge about these processes. However, we can at least describe through which possible mechanisms the DSDs – forcing diversity between the version development processes – may increase failure diversity between the versions. Fig. 2 summarises these possible mechanisms.

The main cause-effect chain is the diagonal series of cause-effect links from "Diversity in human failure in development" to "Failure diversity": causing the intellectual tasks for the two teams of developers to be liable to different mistakes, thus leading to the faults in the different versions, if present, to be likely to be different, which in turn may cause different run-time behaviours, leading to failure diversity. About the practicality of this pursuit, it is sometimes argued that people inevitably tend to make similar mistakes. But psychological research has shown that even substantially identical tasks present greatly varying levels of difficulty depending to the way the tasks are presented. Thus, different design directives (forcing the teams to hold different "natural" views of the development problem, e.g. by partitioning it in different ways), or even different languages and tools,

may shift which parts of the problem are most subject to errors by developers for the different versions.

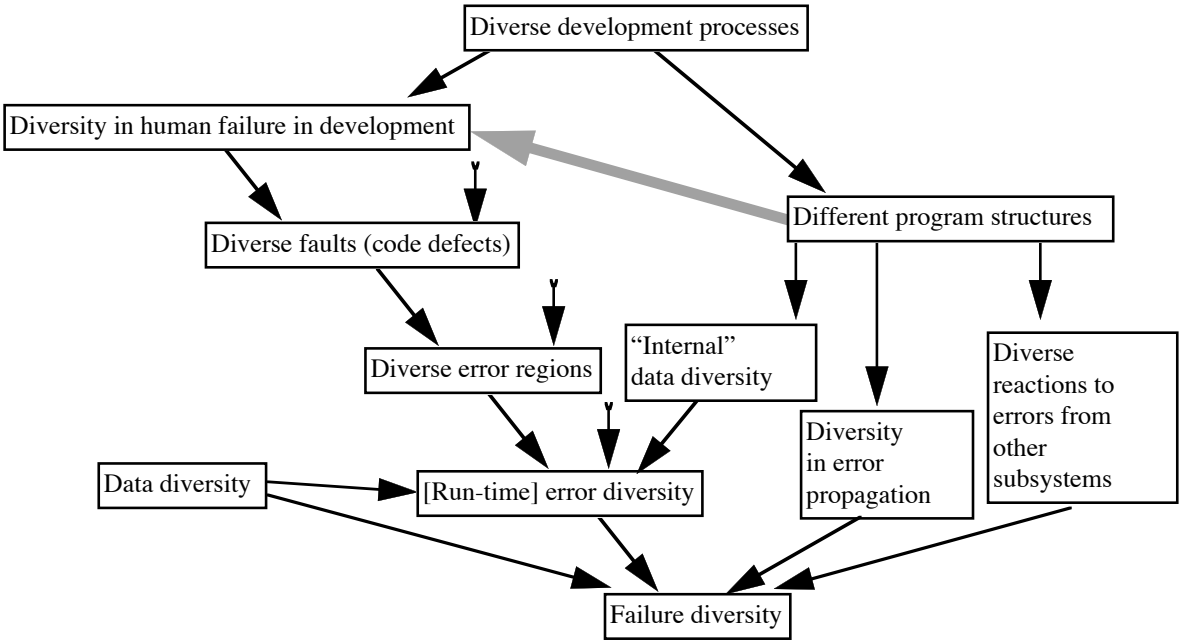

Fig. 2. Mechanisms through which DSDs may improve failure diversity. The arrows indicate causality. The open-tailed arrows indicate that "downstream" differences may occur even without the corresponding "upstream" difference.

In probabilistic terms, the fundamental mathematical model defined by Littlewood and Miller in the 1980s [5] (the "LM model") supports "forcing" diversity via DSDs as a means to reduce the expected value of the system *pfd*. This average *pfd* depends on the *difficulty function* for each version, defined for each possible demand on the system as the probability that the development process will make that demand a *failure point* for that version. The difficulty functions cannot in practice be estimated with confidence, but studying the model clarifies how development processes (and DSDs affecting them) may affect failure independence or correlation. An important implication of this model is, informally, that if two development processes A and B are essentially equivalent in terms of expected reliability delivered, for best reliability in a two-version system *one should choose an "AB" system* (one version developed by process A and one, independently, by process B) *rather than an "AA" or a "BB" system*. An AB system has an expected *pfd* that is at least as good (i.e., low) as that of a homogenous (AA or BB) system, and possibly as low as zero.

Furthermore, the way to reduce the system *pfd*, i.e., to reduce failure correlation between the versions, is to make the two processes differ in such a way that those demands on which versions produced by A are more likely to fail are those in which versions produced by B are *less* likely

to fail (technically, aiming for low covariance between the "difficulty functions" of the two processes [5]).

To link our understanding of the causal description in Fig. 2 to the general directives derived from the probabilistic models, we need to consider how DSDs affect the parameters of the models, specifically the difficulty functions. This is reasonably simple in intuitive terms:

- DSDs that aim to *guarantee separation between the version development teams* attempt to guarantee the condition of independent development, which is necessary for these theorems to apply;
- DSDs that aim to *make the version development processes more diverse* attempt to make the difficulty functions for the two processes more diverse in the specific sense of reducing their covariance. So, DSDs should be chosen with an eye to their effects on *where* (for which demands on the system) each version development process is most "vulnerable".

There is a difficulty with this last recommendation. Available knowledge or expert beliefs about the effects of DSDs usually concern the different likelihoods of each type of software fault (defects in the artefacts – specification, design, code) under different development regimes, not about which demands these errors would affect. The knowledge concerns entities in the code, but the desired result is about their effects in the space of possible demands. A link between the two is missing. There is neither a known mathematical basis nor sufficient empirical evidence for trusting that increasing diversity (between versions) in the faults increases failure diversity [10].

Matching defences to threats

Despite these difficulties in comparing the amounts of gain to be expected from different DSDs, the engineering approach of selecting DSDs, in view of their costs and other problems, on the basis of whether they can plausibly address the specific risk expected, remains valid. The required steps are:

- analysing threats (possible sources of human failures in development, and faults in other systems – development tools and execution platforms – that may cause system failures),
- selecting (on the basis of whatever data are available plus subjective judgement) those against which it is believed that (in the specific project) other defences (before applying diversity) are not strong enough to reduce the risk to acceptable levels, and
- selecting DSDs that are appropriate against these high-risk threats.

The cited report [4] discusses evidence and plausible beliefs about the mechanisms of action of various DSDs, to support this kind of decision process. For instance, a useful coarse-grained classification of threats was between "higher level" errors of the developers (in setting and interpreting high-level requirements and specifications), their "lower level" errors (in

the steps from specification to executable code), and "faults in the support platform" (either run-time hardware and software execution environment, or software tools like compilers). Relatively low-cost DSDs, e.g. hardware diversity, mechanical diversification of code, apply against this last category of threats, but are mostly ineffective against the "higher level" errors. It is commonly believed that current software engineering practices offer better defences against the middle category – "lower level" human error – and, *if these defences are used*, DSDs should be chosen with a focus on the other two categories of threats.

This approach avoids the risk of general, poorly based recommendations of specific DSDs as universally appropriate, and allows instead decisions to be tailored to the needs and circumstances of a project, e.g. the availability of plausible techniques for *avoiding* categories of errors, rather than dealing with their consequences through diversity.

3. Efficacy of DSDs, separation and independence

As pointed out earlier, recommended DSDs try to achieve either better separation between the development processes of different processes (we can then talk about *separation*, or *unforced diversity*) or to make the processes intentionally different (*forced diversity*). Although it is difficult to tailor forced diversity specifically to reduce correlation between version failures – to diversify, between the versions, the sets of demands most likely to be affected by any residual faults – forced diversity gives at least a possibility of these being markedly different, just as unforced diversity gives a possibility of the versions not having identical faults.

It is often asserted that strict separation between the development processes is an essential component of achieving diversity. This would lead to some serious difficulties but is, fortunately, not strictly true. As for the difficulties, let us suppose, for instance, that a version development team reports to the project manager an ambiguity or suspected error in the specification. The project manager will then issue a correction to the specification. Presumably, this correction should go to the other team as well, so as to avoid possible errors. But this violates separation between the teams. Indeed, the way the problem was first noticed may be due to the first team's specific view of a design problem, and the specification change may well transmit this view to the second team, avoiding some errors but also causing this team to share some mental "blind spots" with the first team. This may seem too subtle a risk to consider against the obvious advantage of correcting a wrong specification, but there may be many other reasons for communications that may propagate some influence from one team to the other. Should we then drop the diversity approach altogether? For a high-visibility example, we may consider that Boeing did not use software diversity in the Boeing 777, indicating [12] the need for free communication among developers as the main reason: they assumed that separation between teams was an essential part of the diversity approach. Introducing dependence between the version

development processes also violates the assumptions of the LM model, and thus erodes one of the arguments that recommend "forcing" diversity.

To address these doubts, we reconsidered [8] the claim that diversity requires the strictest separation between version development teams, examining implicit assumptions and plausible arguments and formulating them as precise probabilistic models, which extend the LM model [9]. Of course, some issues cannot be decided by mathematics alone. For instance, when is a specification clarification important enough to be worth transmitting to a development team that did not request it? This question could only be answered empirically, and then only by a volume of experimental research that is probably infeasible. But the probabilistic modelling produces clarity, and it does identify scenarios in which one option in a development process is demonstrably superior to another, in terms of expected system *pfd*.

Our probabilistic reasoning [9] is supported by representing the development process for a two-version system as a *Bayesian network*, as in Fig. 3: in this graph, nodes represent the (a priori unpredictable) choices during a development process, and edges represent the fact that the actual value of the upstream node affects the probabilities of different outcomes for the downstream node.

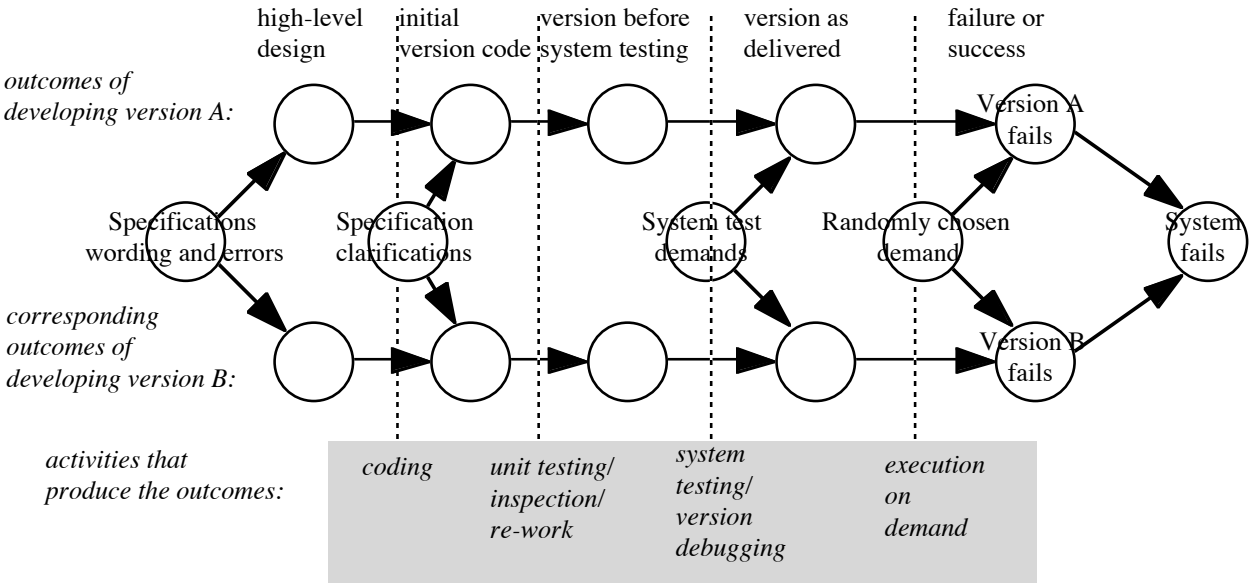

Fig. 3 A Bayesian network for an example of *non*-independent version development processes

We give here some examples of questions that can be answered by our mathematical modelling. A convenient aspect of development for illustrating such examples is testing (*cf* also [7]). At some stage of a project, two software versions (custom-made or off-the-shelf) are

available for testing, in order to correct any remaining faults. Suppose that, up to this point, the diverse development processes are equivalent in that they have the same probabilities of delivering correct programs, or programs with any specific faults. Given a testing criterion, many alternative *test suites* (sets of test cases), all equally appropriate, could be chosen. With respect to testing the two versions, the project manager could decide:

- to randomly choose one test suite and use it for both versions. This creates a form of dependence between the two development process. In the probabilistic model, this dependence produces a *non-negative* addendum in the expression for the expected system *pfd*. However, it reduces other terms in this expression, so that it can be shown that under some broad assumptions this testing stage improves system *pfd*, as one would expect;
- however, this dependence can be avoided by independently selecting, according to the same criterion, *two* test suites for the two versions. Then, the model shows that this is a net improvement: the non-negative addendum disappears.

Similar statements apply to any other "common influence" between the developments of the two versions. What is interesting in the new theorems is that they indicate preferences between alternative decisions for processes between which there is some dependence, and thus are more generally applicable than those derived from the LM model. The two examples given above with reference to testing generalise to two such preference criteria:

- *"Decoupling"* of common influences: given two identical version development processes, with one or more common influencing factors (the nodes between the two chains in Fig. 3), removing any such factor and substituting it with two independent, identically distributed ones each affecting one version can only improve the expected system *pfd*;
- *"Diversifying"* the version development processes: Given the same scenario as above, changing one of the two processes so that its quality remains the same, but the specific probability distribution associated to any one of the nodes changes, can only improve the expected system *pfd*.

A third criterion generalises the first one to the case in which two versions are *not* developed by statistically identical processes:

- *"Decoupling" given positive covariance*: Even if the two processes are not identical, applying "decoupling" will be an improvement if the common factor that is eliminated is one with respect to which the covariance of the difficulty functions is positive, for any possible combination of values of all of the other influences.

An application of this last criterion is a scenario in which the common factor is such that among its possible values (e.g., test suites, algorithms

for implementing a given function, etc.) some are better than (or at least as good as) others *for any demand*: e.g., if of two possible choices we believe that one will certainly imply lower (or at most equal) probability of failure on every demand, *even if we do not know which choice is the better one for our particular application.*

This last form of forced diversity can be applied in a "reactive" manner: after one development team makes a choice, the project manager orders the other team to choose the other alternative. Interestingly, this is, mathematically speaking, a way of creating a *dependence* between the two development processes. That is, the common advocacy of "independent" developments is still justified if all forms of "dependence" create positive correlations between variables in the two processes. If a project manager can devise one that creates *negative* correlation, it will tend to decrease system *pfd*.

4. Combining DSDs

A major problem we mentioned earlier is that of deciding about how to *combine* DSDs. Suppose a certain DSD – one form of diversity between version development processes – is known to be useful, but there are doubts whether it is sufficient protection (i.e., whether it would achieve enough reduction in common-mode failures); suppose that another one is also known to be useful. If we combined the two, would their advantages add up? And if we had a choice between applying two alternative *pairs* of DSDs, how could we decide?

We cannot just assume that applying one more DSD will improve system *pfd*. We could think, for instance, that a DSD (say, specifying diverse algorithms for the various versions) produces benefits because it gives a development team a different 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 "difference" 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, and might even make things worse by forcing the use of inferior techniques (or techniques less familiar to developers), chosen for the sake of diversity.

From the mathematical viewpoint, one of Littlewood's and Miller's theorems (Theorem 3 [3]) gave sufficient conditions for a combination of DSDs to be advantageous (better than applying just one DSD) when applied to version development processes that are *equivalent* (in terms of expected system *pfd*) and *independent*. From the more recent research, we can add some extra theorems that do not require independence. The "decoupling" principles introduced in section 3 apply for any number of independent "influencing factors" that affect both version developments: any DSD that removes one such factor is an improvement.

We have not been able yet to study how broad the application of these decision criteria may be in practice (how often the sufficient conditions of these theorems would be recognised as approximately realised in a development project), but they are certainly worth considering as available guidance.

5. Conclusions

We have summarised some decision problems, and the available knowledge for solving them, in managing diverse developments to produce "diverse enough" versions of the channels of a redundant system. On the minus side, decisions cannot be based on precise quantitative estimates of the *pfd* gains to be expected from the alternative solutions. It is not to be expected that experimental research will deliver complete guidance for these decisions, and some intuitively appealing decision criteria, like trying to diversify the types of errors that are likely to happen in coding, turn out to have very weak support if analysed in detail. Empirical research is still desirable to better understand the causal links in Fig. 2 (via controlled experiments in cognitive psychology and via analysis of diversity in existing code [2; 11], but it will not deliver precise quantitative predictions.

On the positive side, reasoning about the intended and presumable effects of the various "diversity seeking decisions" allows one at least to match them to the expected threats (e.g., errors in specific phases of the development process). In addition, probabilistic modelling, including recent advances in the DISPO projects, helps to clarify the arguments for and against the various possible ways of managing process diversity. In particular,

- it resolves apparent difficulties about the desirability or not of complete separation between the development processes of the diverse versions, and
- it solves some of the difficulties concerning the choice to combine multiple DSDs,

giving clear preferences with respect to some of the decisions that may be necessary.

These mathematical results often confirm recommendations that have so far been offered on an intuitive basis, advocating "separation" between version development processes and "diversification" of these processes. But the mathematical results also clearly outline the limits of these recommendations, for instance highlighting the importance of scenarios in which an action can improve system *pfd* despite reducing diversity, and the role of creating "negative dependence" via "reactive" DSDs. For some procedures, like certain types of testing, the application of these results is straightforward; for others, it may require careful judgement to decide whether the pertinent theorems apply.

A more complete compilation of useful indications for pursuing effective diversity, taking into account the results of recent research, is now being prepared at City University.

Acknowledgments

The work reported was supported in part by the DISPO (DIverse Software PrOject) projects managed by British Energy Generation Ltd, and in part by projects DOTS (Diversity with Off-The-Shelf components) and DIRC (Interdisciplinary Research Collaboration in Dependability) of the Engineering and Physical Sciences Research Council.

References

[1] A. Avizienis. "The Methodology of N-Version Programming", in *Software Fault Tolerance,* (M. Lyu, Ed.), pp. 23-46, John Wiley & Sons, 1995.

[2] I. Gashi, P. Popov and L. Strigini, "Fault Tolerance via Diversity for Off-The-Shelf Products: a Study with SQL Database Servers", *IEEE Transaction on Dependable and Secure Computing*, in print

[3] B. Littlewood and D.R. Miller, "Conceptual Modelling of Coincident Failures in Multi-Version Software", *IEEE Transactions on Software Engineering*, vol. SE-15, no. 12, 1989, pp.1596-1614.

[4] B. Littlewood and L. Strigini. "*A discussion of practices for enhancing diversity in software designs",* DISPO project technical report LS-DI-TR-04, Centre for Software Reliability, City University, 2000.

[5] B. Littlewood, P. Popov and L. Strigini, "Modelling software design diversity - a review", *ACM Computing Surveys*, vol. 33, no. 2, 2001, pp.177-208.

[6] P. Popov, L. Strigini and A. Romanovsky. "Choosing effective methods for design diversity - how to progress from intuition to science", *SAFECOMP '99, 18th International Conference on Computer Safety, Reliability and Security*, Toulouse, France, Springer, September 1999, pp. 272-285.

[7] P. Popov and B. Littlewood. "The effect of testing on the reliability of fault-tolerant software", *DSN 2004, International Conference on Dependable Systems and Networks*, Florence, Italy, IEEE Computer Society, June 2004, pp. 265-274.

[8] K. Salako and L. Strigini. "*Dependence between developments of diverse software versions: scenarios, statistical models and implications",* DISPO2 Project Technical Report LS-DISPO2-04, Centre for Software Reliability, City University, July 2004.

[9] K. Salako and L. Strigini. "*Diversity for fault tolerance: effects of "dependence" and common factors in software development",* DISPO5 Project Technical Report KS-DISPO5-01, August 2006.

[10] L. Strigini. "*DISPO2 fault injection experiments with DARTS software: findings",* DISPO2 Project Technical Report LS-DISPO2-06, Centre for Software Reliability, City University, July 2003.

[11] M.J.P. van der Meulen, P.G. Bishop and M. Revilla. "An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs", *15th International Symposium on Software Reliability Engineering (ISSRE'04)*, Rennes, France, Springer-Verlag, 2004, pp. 101-112.

[12] Y.C.B. Yeh. "Design Considerations in Boeing 777 Fly-By-Wire Computers", *3rd IEEE High-Assurance Systems Engineering Symposium (HASE)*, Washington, DC, USA, IEEE Computer Society Press, November 1998, pp. 64-73.