City Research Online

City, University of London Institutional Repository

Citation: Salako, K. & Strigini, L. (2014). When does "diversity" in development reduce common failures? Insights from probabilistic modelling. IEEE Transactions on Dependable and Secure Computing, 11(2), pp. 193-206. doi: 10.1109/tdsc.2013.32

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/3027/

Link to published version: https://doi.org/10.1109/tdsc.2013.32

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

When does ‘Diversity’ in Development Reduce Common Failures? Insights from Probabilistic Modelling

Kizito Salako, Lorenzo Strigini, *Member, IEEE*

Abstract

Fault tolerance via diverse redundancy, with multiple “versions” of a system in a redundant configuration, is an attractive defence against design faults. To reduce the probability of common failures, development and procurement practices pursue “diversity” between the ways the different versions are developed. But difficult questions remain open about which practices are more effective to this aim.

About these questions, probabilistic models have helped by exposing fallacies in “common sense” judgements. However, most make very restrictive assumptions. They model well scenarios in which diverse versions are developed in rigorous isolation from each other: a condition that many think desirable, but is unlikely in practice.

We extend these models to cover *non*-independent development processes for diverse versions. This gives us a rigorous way of framing claims and open questions about how best to pursue diversity, and about the effects – negative and positive – of commonalities between developments, from specification corrections to the choice of test cases. We obtain three theorems that, under specific scenarios, identify preferences between alternative ways of seeking diversity. We also discuss non-intuitive issues, including how expected system reliability may be improved by creating intentional “negative” dependencies between the developments of different versions.

Index Terms

Common-mode failure, Software Diversity, Fault tolerance, Multiversion software, Probability of failure on demand, Reliability.

I. INTRODUCTION

A defence against design faults in all kinds of systems is redundancy with diversity. In its simplest form, this means that a system is built out of a set of subsystems (known as *versions*, *channels*, *lanes*), which perform the same or equivalent functions in possibly different ways and are connected in a “parallel redundant” (1-out-of-N) or a voted scheme [1]. The rationale for such designs is as follows. If a fault-tolerant design uses multiple, identical copies of a subsystem, all copies contain identical design faults: any circumstances in which one of them were to fail would tend to cause the other copies to fail as well, possibly with results that, despite being incorrect, are plausible, consistent and thus cannot be recognised as failures. Diversity eliminates the certainty of design faults being reproduced identically in all channels of the redundant system. One can hope that any faults (rare, given good quality development) will be unlikely to be similar between channels, causing them not to fail identically in exactly the same situations. This low probability of common faults can be sought by seeking “independence” between the developments of the multiple versions. For instance,

- for custom-developed components, development teams work separately, making their separate design choices (and possible mistakes), within the constraints of the specifications and general project management directives. These directives may also be specifically geared at “forcing” more diversity, e.g. mandating different architectures or different development methods [3], [4], [5].
- when building the diverse channels out of pre-existing components, one can seek assurance that the developments were indeed separate and, for instance, did not rely on common component libraries or designs.

We study a scenario of diversity between *software* versions, the application of diversity that is most discussed in computing, though our modelling approach applies more generally.

Difficult questions for a system designer or project manager concern how effective the various ways of pursuing diversity are towards reducing common failures, so that one can decide when to use diverse designs, and how to manage their procurement. These questions have generated lively debates, in which positions are often supported by informal appeals to experience and individual judgement. As usual in software engineering, experimental evidence is limited and hard to generalise, especially to high-reliability systems. Here we attempt a rigorous probabilistic description of the issues involved, to provide insight, clarify

---

K. Salako and L. Strigini are with the Centre for Software Reliability, School of Informatics, City University London, Northampton Square, London EC1V 0HB, UK (telephone:+ 44 20 7040 {0112, 8245}, e-mail: {kizito, strigini}@csr.city.ac.uk).

Manuscript received January 07, 2013; revised July 25, 2013.

[1] A *parallel redundant (1-out-of-N)* system is one in which correct system functioning is assured, if at least one channel functions correctly. In a *voted system*, correct system functioning is assured if a majority of channels function correctly. Many other architectures using diversity are possible [1], [2].

assumptions and claims made in this debate, and separate the questions that can only be answered empirically from those that can be answered by deduction. Among the latter, we show general conditions under which some of the above-mentioned ways of seeking “independence” are guaranteed to improve expected system reliability.

A. Background: effectiveness of diversity

A basic question is: what probability of common failure can we expect for a system of two or more, “independently” developed versions? Statistical independence between their failures is often seen as the ideal result to be sought. It would support assurance of very high reliability of the redundant system at relatively low cost. For instance, we could trust that a safety system, implemented as a 3-version parallel system, has a *probability of failure on demand*[2] (*pfd*) of no more than 10^{-6} at the very affordable cost of demonstrating that each version has a *pfd* of no more than 10^{-2}. This is an attractive proposition, since trying to demonstrate very high reliability of software imposes high costs for generally limited returns in terms of confidence [8], [9]. An even better scenario would be one in which common failures never happen.

In reality, one can say very little, *a priori*, about the probability of common failure for a *specific* pair of versions. Some pairs may have no fault that causes them to fail on the same demand. In some other pairs, every time one version fails the other one will fail as well (*cf.* section II).

Could we at least predict something about the *average* results of applying diversity in a certain system? Will the “average pair” of versions behave like a pair of two “average versions” failing independently? A famous experiment by Knight and Leveson [10] refuted this conjecture: this “independence on average” property did not hold in the specific population of versions that their subject programmers developed, and thus cannot be assumed to hold in general. On average, pairs of versions failed together with far higher probability than the square of the average *pfd* of individual versions (though far less frequently than the average individual version). This suggested that the general law in diverse-redundant systems may be, unfortunately, one of positive correlation – on average – between version failures. Experimental evidence was not enough to support or refute such general claims; probabilistic modelling offered an explanation of how such correlation may arise. The seminal models were due to Eckhardt and Lee [11], and Littlewood and Miller [12] (We will refer to them as the “EL” and “LM” models). The bases of the EL model were:

- from the viewpoint of reliability, a program can be completely described by its behaviour – success or failure – on every possible demand;
- the process of developing a program is itself subject to variation. One cannot know in advance exactly which program will result from it and, specifically, which faults it contains (even after delivery, the code may be known, but not, crucially, on which demands it fails). Development can be modelled as a process of *random sampling* which selects one program from the population of all possible programs. The visible properties of the process (system specifications, methods used, choice of developers) do not determine *exactly which program* is created, but they determine the *probabilities* of it being any specific one;
- some demands are more difficult for the developers to treat correctly than others. One can formally model this “difficulty” of each particular demand via the probability of a delivered program, “randomly” chosen by the development process, failing on that specific demand.

In this modelling framework, the reputedly ideal condition of complete isolation between the developments of the various versions is represented by the assumption that each program version is selected (sampled) *independently* of the selection of any other. Eckhardt and Lee [11] showed that if two versions are produced by identical development processes then, even if the two processes are independent, still the versions produced cannot be expected to fail independently; and “independence on average”, rather than the norm, would be a limiting, best case scenario. Later, Littlewood and Miller [12] pointed out that the versions may be developed by *different* processes: this is indeed the purpose of “forcing” diversity. Then, the “correlation on average” between failures of the versions could be negative as well as positive. This could even lead to zero expected system *pfd*, even if both versions were likely to be faulty.[3] Both the *EL* and *LM* models bring important insights:

- *even perfect isolation* –and thus independence – between the developments of the versions does not imply an expectation of independence between their failures. Independent developments are indeed naturally modelled by the property that, given a specific demand, two – independently “sampled” – versions will fail independently, *conditionally on* that demand, and yet this in turn implies non-independence for a randomly chosen demand;
- “forcing” diversity may avoid this unwelcome result;
- furthermore, we get a clear, formal description of conditions that increase failure dependence, and thus of which goals we should pursue when we try to “force” diversity.

---

[2] The *pfd* is a common measure of reliability for systems that work on a per-demand basis. A *demand* is a discrete stimulus to which a system is required to respond. A demand can be as simple as a single invocation of a re-entrant procedure, or as complex as the complete sequence of inputs to a flight control system from the moment it is turned on before take-off to when it is turned off again after landing. The same modelling approach can be extended, with some extra complexity, to continuously operating software, to model reliability as a function of continuous time [6], [7].

[3] These are *possible* results of “forcing” diversity. There will be *some* difference between the development processes of different versions, so the LM assumptions always apply (given isolated developments); but this fact alone does not justify an expectation of especially low system *pfd*.

These implications have been explored in various applications of this modelling approach, e.g. to the choice of fault removal methods [13], security [14], and human-machine systems [15].

B. *Open questions about development of diverse software*

Both the EL and LM models include what we call an "independent sampling assumption" (ISA), representing an extreme scenario of perfectly isolated developments of the versions. The ISA allows elegant theorems like the EL model's implication of "positive correlation on average".

Here, we study the consequences of violating this assumption, to extend the range of practical insight offered by this kind of probabilistic models. There are at least two reasons for seeking this extension. The first reason is that "perfectly isolated" developments appear unlikely in practice. Avoiding all information flows, even indirect, between the version development teams seems hard; developers often share common education background, or use the same reference books, etc.; normal actions like clarifications and amendments to the specifications will affect all the development teams.

These scenarios prompt several questions as to their practical implications: do they violate the ISA? And if so, is it still true that failure independence is unlikely and it is prudent to assume positive correlation? And do they invalidate any other practical guidance drawn from the EL and LM models?

The second reason for studying violations of the ISA is that the importance of the EL model is in the warning: "Even under the most optimistic assumptions – perfect independence in development – still, given identical processes, you should expect positive correlation of failures". But what if the ISA were not the "most optimistic" assumption? The debate about desirable practices in diversity has seen over the years many claims that are still undecided. In particular,

- some authors [3] emphasize the goal of "independence" between the development of diverse versions, by strict isolation of the development teams, from each other (to avoid the propagation of mistakes between the developers of different versions – "*fault leaks*") and from all unnecessary common influences on them;
- others similarly posit that strict isolation of developments is necessary for effective diversity, but argue that it will impede communications within a system development team, *impairing* system dependability. This is the argument given for the decision *not* to use application software diversity in the Boeing 777 [16].

But this last argument suggests an all-or-nothing choice: complete isolation, or no diversity at all. This is instead a matter of trade-offs. For instance, correcting a specification error removes a *likely* cause of faults in more than one version; but researchers have worried [3], [4], [17] that it may introduce another *possible* cause of common faults: e.g., a specification update reflecting a question by developers could propagate to other developers the same view of the implementation problem. That is, commonalities in development may reduce the probability of any one version failing, but could increase the conditional probability that if one fails, others may fail with it. System reliability depends on both these probabilities, so there is a trade-off, and perhaps the best compromise is not always complete isolation: on average, diverse developments made "less diverse" by communication might be better than either complete isolation or a single-version system.

Thus, the ISA might not be a "best case" assumption. What changes, then, in the advice for practitioners? Does the pessimistic warning from the EL model become invalid? How should developers of systems with diverse redundancy find a good compromise between the advantages of isolation and of communication? To help with such decisions, we need to separate principles that can be justified *a priori* on mathematical grounds (starting from assumptions that seem empirically justified, or anyway trusted with some confidence) from those that can only be judged on empirical data. For the former, we aim to state clearly their meanings and the premises under which they hold.

In the rest of this paper, Section II introduces the reference scenario for the discussion and some terminology, mostly from the previous literature [12], [11], [7]. Section III recalls the previous theory (EL and LM models) and introduces Bayesian Belief Networks to represent the stochastic models we use. Section IV analyzes how causes of common failures relate to violations of the "Independent sampling assumption" (ISA), and their implications for expected system reliability. Section V presents our novel "preference criteria" between system development processes, regarding intentional separation and diversification of these processes. Section VI discusses the practical implications of these mathematical considerations for decisions about managing a multiple-version development process and whether these change the lessons drawn from the EL/LM models. Last, we summarise our results and indicate directions for future work.

II. REFERENCE SCENARIO AND TERMINOLOGY

We consider a system that may be implemented either as a single version or as a diverse 2-channel, 1-out-of-2 system (Fig. 1). This simplest of diverse-redundant architectures has important concrete applications (e.g., many safety systems); it can be described by simple models, and yet it presents the essential difficulties of all diverse-redundant systems.

We use the term "[program] versions" to mean diverse, equivalent implementations of the system functions. [4] Following common usage, when there is no risk of ambiguity, we also call "version" one of the diverse channels of the two-version system. We avoid the other meaning of the term "versions", i.e., "results of successive changes to a program", "releases".

---

[4] This equivalence may be with respect to functions defined at a very high level. E.g., a function might be "emergency shutdown" of a physical plant, for which the diverse "versions" may include sensors and actuators applying different physical principles of action, and diverse computer hardware running different algorithms. Thus, our terminology and basic models are meant to cover quite a broad range of systems.

TABLE I
SYMBOLS AND ABBREVIATIONS

Symbol	Definition
pfd	Probability of failure on demand.
$r.v.$	random variable.
x	Value of a demand on a system.
$\mathcal{X}$	The set of all possible demands.
X	The next demand occurring in operation (a random variable).
$\theta\left(x\right)$	*Difficulty function*: probability of a randomly selected software version failing on demand x
π	A program, a software version.
Π	A r.v. representing the development of a program as a random selection
$\omega\left(\pi, x\right)$	*Score function*: indicator function of whether software version π fails on demand x.
$P\left(Y=y\right)$	Probability of r.v. Y taking on value y.
$P\left(Y=y \mid W=w\right)$	Conditional probability of the r.v. Y taking on value y, given that the r.v. W takes on value w.
$\mathbb{E}\left[Y\right]$	Expectation of the random variable Y.
$f_{A;y}(z)$	A function (associated with channel A) of the realisations of the random variables Y and Z.
$\mathbb{E}\left[Y \mid Z=z\right]$	Conditional expectation of the r.v Y, conditional on the value z of the r.v. Z.
$\mathbb{E}\left[\mathbb{E}\left[Y \mid Z\right]\right]$	Expectation of the random variable $\mathbb{E}\left[Y\vert Z\right]$. This evaluates to $\mathbb{E}\left[Y\right]$.
$\mathrm{Var}\left[Y\right]$	Variance of the random variable Y.
$\mathrm{Cov}\left[Y, Z\right]$	Covariance of the random variables Y and Z.
$\mathrm{Cov}\left[Y, Z \mid W=w\right]$	Conditional covariance of the r.v.s Y and Z, conditional on the r.v. W taking the value w.
Note: in each definition, *random variable* may be replaced with *random vector*	

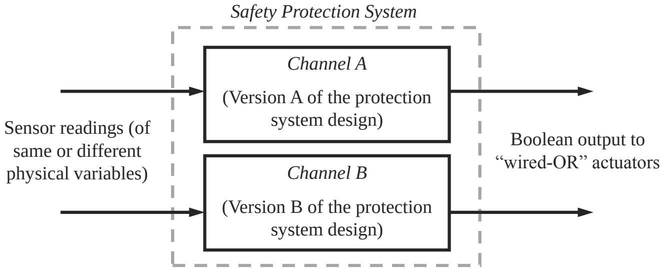
Safety Protection System
Channel A
(Version A of the protection system design)
Channel B
(Version B of the protection system design)
Sensor readings (of same or different physical variables)
Boolean output to "wired-OR" actuators

Fig. 1. Our reference 1-out-of-2 system, an abstraction of a plant safety system with two diverse channels. A demand is a sequence of inputs that indicates the plant entering a potentially hazardous state; successful processing of a demand is the triggering of plant shut-down. The outputs of each channel, and of the system, are logically Boolean variables.

We use this scenario of multiple-version development:

- each version is produced by its *version development team* (perhaps further divided, as usual, into sub-teams for design, coding, inspection, testing, etc.);
- one *project manager* or "manager" defines the requirement specifications that the development teams must implement and the constraints under which they have to work, issues specification updates and decides on final acceptance of the developed versions;
- each version is developed by a *version development process*: the set of conditions affecting development, like the members of the development team, the specification they are given to implement, the development methods they use (including the verification and validation policy), any design details that are specified for them *a priori*, the schedule and budget constraints, etc;
- we call *system development process* the combination of the development processes for the two or more versions in a system, plus the way these processes are co-ordinated.

If the versions are built from off-the-shelf components rather than from scratch, the same terminology applies, with the only difference that the "project manager" has less minute control on development, only "choosing" the versions' development processes indirectly, by selecting their products.

Other aspects of our reference scenario are:

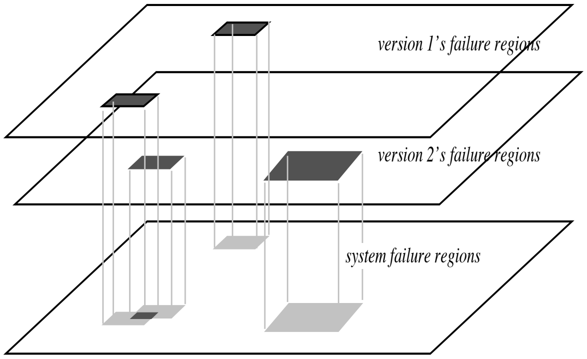

Fig. 2. Overlaps between the failure sets (sets of demands that cause failure) of two diverse software versions. Each horizontal, rectangular surface represents the set $\mathcal{X}$ of all possible demands on the system. The shaded rectangles on the three surfaces depict the failure sets of the two versions, and that of the 1-out-of-2 system – the intersection of the two.

- we consider an "on demand" system. It receives demands from its environment, and the result of processing a demand is either a success (a correct response) or a failure (an incorrect response) by the system;
- the dependability measure of interest is the system's *probability of failure on demand* (*pfd*). For our purposes, the nature of the required response to a demand – e.g., whether it is turning on an alarm signal or controlling complex mechanical actuators – is irrelevant. We only distinguish between two types of response to a demand – success (i.e. correct behaviour), and failure;
- we only consider "systematic" failures, due to design faults, not covered by the usual analyses methods for "random" failures.

During execution, there is uncertainty about which demand the software will next receive from its environment. This next demand is thus a random variable X, taking on values in the set of all possible demands, $\mathcal{X}$. A probability distribution, the *demand profile*, specifies for each demand $x \in \mathcal{X}$ its probability of being the next demand the software receives.[5] The demand profile summarises, and depends on, the circumstances in which the system is used. It will generally be known with some degree of imprecision. We use the phrase "a randomly chosen demand" to mean a demand that occurs according to this random process. We assume the demand profile as fixed and given. The system's *pfd* is the probability of the system failing on a demand X, randomly chosen according to this specific demand profile.

Each version may contain faults, produced by the uncertain and variable process of software development. Due to its faults, a version fails deterministically on certain demands, its *failure set*.[6] The sum of the probabilities of these demands is the *pfd* of that version: a demand profile associates a *pfd* value to the failure set of a version. The system's failure set is the intersection of the versions' failure sets (Fig. 2).

III. BASIC CONCEPTUAL MODELS OF THE DEVELOPMENT AND FAILURE OF DIVERSE-REDUNDANT SYSTEMS

In this section we briefly recall the EL and LM models [11], [12], which we extend in section IV-B.

A. Describing failure probabilities, given known failure sets

Given a program (or system) that behaves deterministically, we can characterise its faults via a Boolean *score function* $\omega(\pi, x)$, defined for each demand, x, and program, π, as:

$$\omega(\pi, x) = \begin{cases} 0, & \text{if } \pi \text{ processes } x \text{ correctly} \\ 1, & \text{if } \pi \text{ fails on } x \end{cases}$$

Score functions are normally unknown, but they are a useful device for reliability modelling.

For the next randomly chosen demand X, the score function of π, $\omega(\pi, X)$, is also a random variable ("r.v." henceforth), and π's *pfd* is its expected value:

$$\begin{aligned} \mathit{pfd}_\pi &:= P\left(\omega\left(\pi, X\right) = 1\right) \\ &= \mathbb{E}\left[\omega(\pi, X)\right] \end{aligned} \tag{1}$$

where (*cf.* Table I), "$\mathbb{E}\left[\omega(\pi, X)\right]$" is the mean of the r.v. $\omega(\pi, X)$, with respect to the distribution of the r.v. X.

[5] We follow the convention of using uppercase letters (e.g. X) for random variables, and lowercase letters (e.g. x) for the values which they can take.

[6] The assumption of determinism can be relaxed. This implies making the ω function (see section III-A) a probability rather than a binary value.

B. Describing uncertainty about the failure points: version sampling distribution and difficulty function

Modelling program development as random sampling, we define a r.v., Π, whose realisation is the specific program that is in the end produced. Π will take on a value π from the set of *all possible programs*[7].

The "version development process" of a program (section II) determines a *version sampling distribution*: the distribution of Π. That is, for each program version π that could be produced, the *version sampling distribution* specifies the probability $P\left(\Pi=\pi\right)$ of π actually being produced.

This way of modelling acknowledges the variability of the outcome of a software development process. Even a closely controlled process does not strictly determine the software produced, with its faults. We expect different "values" of the circumstances of development to induce different distributions of the r.v. Π. The development of a program under given circumstances is modelled as running the stochastic version development process once, "extracting a program at random" from a population of programs. We will use phrases like "a randomly selected program [version]" in this sense: the program may be yet to be built, or already delivered, but it is still unknown in that its score function is unknown.

We call *difficulty function* $\theta(x)$ ([12]) the function whose value on demand x is the probability of a "randomly selected" software version failing on x. For brevity, we call the value of a *difficulty function* on a demand the *difficulty* of that demand. Each program π has an associated score function, $\omega(\pi, x)$, defined on the space of demands, $\mathcal{X}$. For a demand x, given a version sampling distribution, the difficulty of x is:

$$\begin{aligned}\theta\left(x\right) \quad &:= \quad P\left(\,\omega\left(\Pi, x\right)\,=1\,\right)\\ &= \quad \mathbb{E}\left[\omega(\Pi, x)\right]\\ &= \quad \sum_{\mathrm{all}\pi}\omega\left(\pi, x\right)P\left(\Pi=\pi\right)\end{aligned} \tag{2}$$

The difficulty function depends on the version development process and thus on the development team. Intuitively, the more likely it is that a version will be developed which fails on the demand, the more difficult the demand is.

Distinct versions may have different *pfd*s. The average *pfd* over all possible versions is the probability of a randomly chosen version failing on a randomly chosen demand:

$$\begin{aligned}\mathbb{E}\left[pfd\right] \quad &:= \quad P\left(\,\omega\left(\Pi, X\right)=1\,\right)\\ &= \quad \mathbb{E}\left[\theta\left(X\right)\right]\end{aligned} \tag{3}$$

C. Implications for two-version system

The EL and LM models imply that, for two software versions, even if independently developed, it is wrong to estimate their joint *pfd* by multiplying their individual *pfd* estimates. The reasoning is as follows. For a specific system development project (with its particular constraints and circumstances), the two redundant channels, A and B, have associated r.v.s, Π_A and Π_B, representing the program version that will eventually be produced for each channel. Π_A and Π_B have associated version sampling distributions, and thus difficulty functions, $\theta_A(x)$ and $\theta_B(x)$. In the EL model, the development teams develop their versions "in similar ways", to the extent of "selecting them randomly" according to the *same distribution*, though *rigorously independently*. For any given demand, they are equally likely to make mistakes on that demand – they have identical difficulty functions, $\theta_A(x)=\theta_B(x)=\theta(x)$. When this randomly chosen pair of versions receive the same, randomly chosen demand, the probability of both failing is:

$$\begin{aligned}P\Big(\omega\left(\Pi_A, X\right)\omega\left(\Pi_B, X\right)\,=1\Big) \quad &= \quad \mathbb{E}[\omega(\Pi_A, X)\omega(\Pi_B, X)]\\ &= \quad \mathbb{E}[\theta_{AB}(X)] \;=\; \mathbb{E}[\theta_A(X)\theta_B(X)]\\ &= \quad P\Big(\omega\left(\Pi_A, X\right)\,=1\Big)P\Big(\omega\left(\Pi_B, X\right)\,=1\Big)+\mathrm{Var}\left[\theta(X)\right]\\ &= \quad \mathbb{E}\left[\,pfd_A\right]^2+\mathrm{Var}\left[\theta(X)\right]\end{aligned} \tag{4}$$

where $\theta_{AB}(x)$ is the *joint* (or system) difficulty function, defined similarly to the difficulty functions for single versions. Since the versions are developed independently, $\theta_{AB}(x)=\theta_A(x)\theta_B(x)$ (see Appendix A). In (4) we have used the fact that for identical version sampling processes $P\left(\Pi_A \text{ fails on } X\right)=P\left(\Pi_B \text{ fails on } X\right)$; $\mathrm{Var}\left[\theta(X)\right]$ is the variance of the r.v. $\theta(X)$ and thus (as for any r.v.) is non-negative: (4) shows that the expected system *pfd* can be no better than the square of the expected *pfd* of a single version, which would be the expected system *pfd* if the versions failed independently. *The problem is the variation of difficulty* across the demand space: $\mathrm{Var}\left[\theta(X)\right]$ is zero only if (implausibly) the difficulty function does not vary between demands.

---

[7] This set is large but finite. In the broadest sense, it includes any sequence of zeros and ones that fits in the finite computer memory available. Many such sequences have zero probability of being produced, as they are meaningless or implement functions completely different from that specified.

The LM model instead recognises that the version development processes are unlikely to be identical: in general, $\theta_A(x) \neq \theta_B(x)$, at least for some x. In particular, the project manager can *impose* different constraints on the two version development teams ("forced diversity"). So, unlike in EL, the expected system *pfd* is:

$$P\Big(\omega(\Pi_A, X)\,\omega(\Pi_B, X) = 1\Big) = \mathbb{E}[\mathit{pfd}_A]\ \mathbb{E}[\mathit{pfd}_B] + \mathrm{Cov}[\theta_A(X), \theta_B(X)]\ . \tag{5}$$

The covariance term could be negative, so that the average system *pfd* could be better than the product $\mathbb{E}[\mathit{pfd}_A]\ \mathbb{E}[\mathit{pfd}_B]$: the lower bound for the mean system *pfd* is no longer this product term. The average system *pfd* could even be 0, even if $\mathbb{E}[\mathit{pfd}_A]\,\mathbb{E}[\mathit{pfd}_B] > 0$ (*iff* on every x at least one between θ_A and θ_B has value 0). So, the covariance term is a measure of how often demands on the system are demands for which the two versions' difficulty functions are both especially low, or high. *The effectiveness of "forcing" diversity depends on the extent to which it makes the demands that are "more difficult" for one version "less difficult" for the other one.*

D. Representation of the probabilistic models via BBNs

The EL and LM models highlight the importance of conditional independence relations between random events when reasoning about the probability of coincident failure between software versions. To describe such relations, *Bayesian networks* (or "Bayesian Belief Networks", *BBNs*), as in Fig. 3, are a convenient formalism; they:

- depict conditional independence relations (and, thus, joint distributions) between random events
- can be analyzed, by rules applied to the topology of the BBN, to elicit the consequences of conditional independence relations between sets of random events.

We introduce basic concepts with reference to Fig. 3. References for the mathematical underpinnings of BBNs [18], their applications [19] and their uses in dependability models [20] are listed in the Bibliography.

A BBN is a directed, acyclic graph. Each node represents a random variable. Nodes without common parents are mutually independent r.v.s. Nodes with common parents are independent, *conditionally* on the values of the *common* parent nodes. For each node, a set of conditional probability distributions is defined: the distributions of the r.v. associated with that node, conditional on each possible combination of values of the r.v.s associated with the node's parents. For instance, in Fig. 3 the probability distribution of node Π_A is the version sampling distribution of the version development process for system channel A; the distribution associated with the binary r.v. "Π_A fails" is, for each pair of values $\{\pi, x\}$ of the r.v.s Π_A and X, given by the score function $\omega(\pi, x)$. The probability distribution of the binary r.v. "*System fails*" is a deterministic function (a degenerate probability distribution): for all combinations of values of the binary variables "Π_A fails" and "Π_B fails", "*System fails*" equals their product with probability 1.

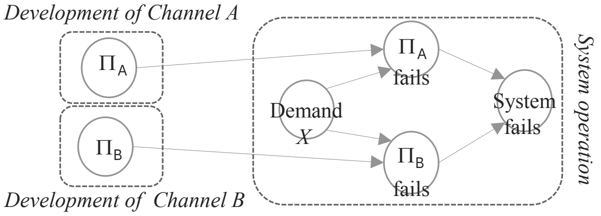

Fig. 3. Bayesian network (BBN) representing the EL and LM models. The absence of parent nodes common to Π_A and Π_B represents the assumption that the versions are chosen independently (ISA). The single common ancestor X for the two nodes "Π_A fails", "Π_B fails" shows that they fail independently, *conditionally* on the randomly chosen demand X. The dashed-contour, rounded boxes (not part of BBN syntax) identify subgraphs that model distinct physical processes.

A BBN's topology implies certain conditional independence relations among its nodes. Two r.v.s in a BBN, say X and Y, are conditionally independent, conditional on some set of r.v.s Z, if every path (consecutive sequence of edges and nodes) between X and Y is *blocked* by Z. In this case we say that Z *d-separates* X and Y. A path, p, is said to be blocked by Z if and only if

- p contains a chain $i \to m \to j$ or a fork $i \leftarrow m \to j$ such that $m \in Z$, or;
- p contains an inverted fork (or collider) $i \to m \leftarrow j$ such that $m \notin Z$ and no descendant of m is in Z.

If Z does not *d-separate* X and Y, and all the nodes/elements of Z are ancestors of Y, say, then it is possible to marginalise (calculate an average of) the joint distribution of Y and Z to obtain the distribution of Y. This important transformation allows us to transform seemingly different BBN models of system development processes into a form (Fig. 6) that highlights sources of probabilistic dependence between the developments, and failures, of diverse versions.

In summary, conditional independence can be captured by the property of *d-separation* between the nodes of some suitable BBN topology. To analyze relationships between random events in a system's lifecycle, we can use a BBN representing the joint probability distributions of r.v.s that model these events. For example, in Fig. 3 the nodes Π_A, Π_B and X are the only parent nodes of "Π_A fails" and "Π_B fails", which in turn are the only parents for "System fails". This shows that whether the system fails on a demand is determined by which versions form the system, and by the demand.

IV. COMMONALITIES BETWEEN DEVELOPMENT PROCESSES; VIOLATIONS OF "INDEPENDENT SAMPLING"

A. "Independent sampling" and its possible violations

The Independent Sampling Assumption (ISA) of the EL and LM models describes an ideal regime of complete separation between the developments of two versions: with "perfect" separation, there is no way that the development of one version may influence the development of another one. There is *conditional independence*, given any specific demand x, between the failures of the two versions: the probability of x being a failure point for one version does not depend on whether it is a failure point for the other version; or, the probability of building a two-version system that fails on x is the product of the probabilities of each version development team building a version that fails on x: $\theta_{AB}(x) = \theta_A(x)\theta_B(x)$. Thus, the ISA allows one to derive equations (4) and (5) (see Appendix A).

Even with this restrictive assumption, the EL and LM models can model many "commonalities" that are of concern in developing or procuring diverse systems. For instance, a concern may be that different development organisations share a culture – staff educated by similar schools, using similar reference books, etc – and thus are liable to the same "mental blind spots" in their work. These shared blind spots are modelled via "high difficulty" sets of demands, without violating the ISA: similarity of culture affects the developers even though their activities are otherwise isolated.

There are, however, several reasons for studying scenarios in which the ISA is false:

- complete separation is unlikely for various practical reasons. So, we ought to study the effects of the inevitable, though possibly small, departures from it;
- communication between the teams may sometimes be desirable, in situations in which:
  - despite increasing correlation between failures of the versions on each demand, it improves the reliabilities of the individual versions so much that the net effect is improved system reliability;
  - or it is devised so as to *reduce* correlation between failures of the versions and improve system reliability.
- even without letting the teams interact, the project manager may wish to improve the expected *pfd* of the diverse-redundant system by methods that could violate conditional independence. Examples are:
  - allowing the two teams to choose freely the algorithms for implementing the specification, but with the constraint that they use different algorithms The hope is to produce negative correlation between the team's mistakes on the same demand (*cf* App. E) ;
  - mandating some common quality assurance procedure which may cause *positive* correlation. For instance, testing the two versions on the same test cases may be a cost-effective way of pursuing reliability of both versions and thus of the system.

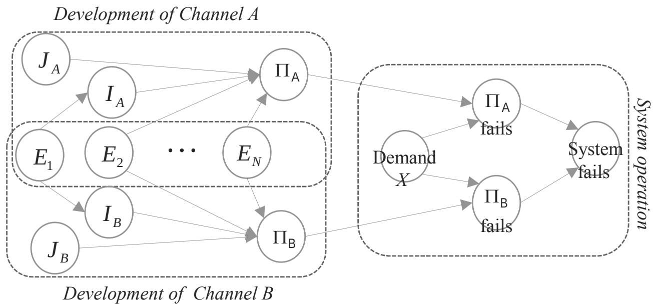

Fig. 4. A BBN depicting a (two-version) system development process affected by multiple influences. The nodes to the left of Π_A and Π_B may represent, e.g., design artefacts, test techniques and test cases, and factors affecting these various aspects of development, like communication between the teams and the project manager. Influences may interact, e.g., errors in a specification may affect choices of test cases, and both affect which version is delivered. Some influences, like J_A or I_B, affect the development of only one version; others – $E_1, \ldots, E_N$ – affect both.

B. Modelling violations of "independent sampling"

To describe situations in which the version developments are not completely isolated from each other we introduce the concept of [random] *influences*: any events or circumstances that affect a development project and may vary unpredictably, modelled as r.v.s (or random vectors). This randomness can represent "uncertainty in the world" (*aleatory* uncertainty), if we wish to predict the results of a development process that has yet to happen and is affected by random factors. However, it can just as naturally represent "uncertainty in knowledge" (*epistemic* uncertainty): the development has taken place, but what we know about it does not include the values of all these variables. Influences may represent events external to the software development process (e.g. requirements or budgets) or generated internally (e.g. choice of test cases).

We are interested in the effects of *common influences*: those that affect the developments of both versions. We will show that such common influences indeed affect (increase *or* reduce) correlation between version failures. Figs. 4 and 5 show two scenarios with multiple influences, respectively a generic process and one in which a set of concrete common influences and r.v.s are identified.

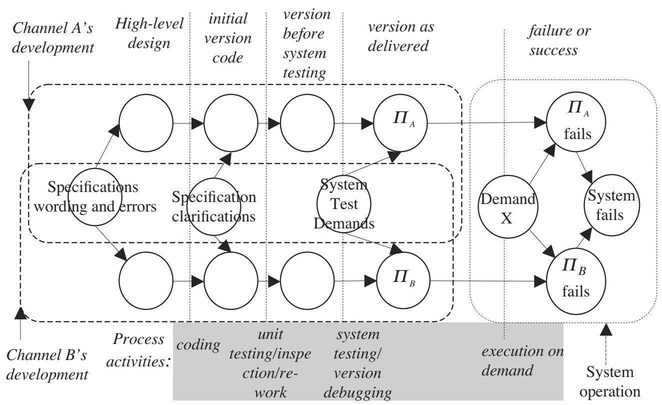
Channel A's development
High-level design
initial version code
version before system testing
version as delivered
failure or success
Π_A
Π_A fails
Specifications wording and errors
Specification clarifications
System Test Demands
Demand X
System fails
Π_B
Π_B fails
Channel B's development
Process activities:
coding
unit testing/inspection/re-work
system testing/ version debugging
execution on demand
System operation

Fig. 5. A BBN with examples of common influences between the developments of two versions. The top and bottom rows of nodes represent artefacts at successive stages of production of the two versions. Their names are listed above the graph.The labels under the graph name the activities that transform an artefact into the next one. Each is subject to some degree of randomness in its results, justifying the views of each artefact as a random variable, whose distribution is determined by the values of its parent nodes: the artefact "upstream" (to the left) of it, and in most stages a common influence as well: a node in the middle row. Here, only the unit testing phase is performed without common influences affecting both teams.

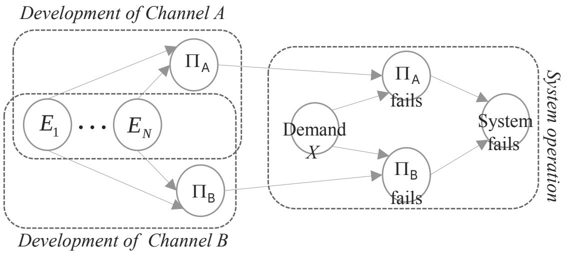
Development of Channel A
Π_A
E_1
E_N
Π_B
Development of Channel B
Π_A fails
Demand X
System fails
Π_B fails
System operation

Fig. 6. The BBNs in Figs. 4 or 5 can be transformed into a shape like this BBN: the only influences are direct "parent" nodes of Π_A and Π_B, common to both version development processes. The intermediate nodes through which the influences affect Π_A and Π_B have been removed by marginalising the distributions in the BBN of Fig. 4 with respect to the non-common influences. Thus, many scenarios in which the ISA does not hold can be described by the same form of equation that applies to this figure, and which we study in this paper.

If the value of an influence is known, we can stop seeing it as a r.v. and can remove it from the BBN. The BBNs for the EL and LM models (Fig. 3), contain only the right-hand part of the BBNs in either Fig. 4 or Fig. 5, meaning that common influences are either absent or considered known.

If in a development process the sampling distribution depends on the values $e_1, \ldots, e_N$ of N random influences, we can still talk about a difficulty function, the probability of a randomly sampled program failing on a certain demand. For a given actual history of a development project, among the many possible histories, there will be a difficulty function conditional on the specific values that all influences actually took[8]:

$$\begin{aligned}\theta_{e_1,\ldots,e_N}(x) &:= P\left(\omega(\Pi, x) = 1 \,|\, E_1 = e_1, \ldots, E_N = e_N\right) \\ &= \sum_{\text{all } \pi} \omega(\pi, x)\, P\left(\Pi = \pi | e_1, \ldots, e_N\right) \ . \end{aligned} \tag{6}$$

Averaging this conditional difficulty function, over all the possible combinations of values of the influences, gives the *marginal* difficulty $\theta(x)$ – the probability of demand x being a failure point, given all the possible histories of the development process. The conditional difficulty is a r.v., since it is a function of the common influences, which are r.v.s: we use the notation $\theta_{e_1,\ldots,e_k,E_{k+1},\ldots,E_N}(x)$ for a difficulty function conditional on specific values, $e_1, \ldots, e_k$, of some influences, for a range of possible development histories in which the remaining influences, $E_{k+1}, \ldots, E_N$, take on values according to their respective probability distributions.

In general, these various difficulty functions differ. E.g., the event "demand x will be a failure point in the delivered software" has different probabilities depending on whether an influence value is given, say "specification change 4.3.12 was issued, clarifying required behaviour on x" or no value is given – it is not known whether any such clarification will be issued.

We can transform a BBN like that in Fig. 4 (or Fig. 5) into a form in which the only influences are common influences, such as Fig. 6, by averaging (marginalising) over those nodes (r.v.s) in Fig. 4 that we remove to obtain Fig. 6. This transformation preserves the meaning of the original BBN in Fig. 4: it implies no conditional independence assumptions that were not already implied by the original BBN; and all the joint distributions between the nodes in Fig. 6 are unchanged from those between the homologous nodes in the original BBN [18]. However, Fig. 6 omits the details of how the non-common influences affect the development process (e.g. which phase of development they affected). [9]

---

[8] The version sampling distribution $P(\Pi = \pi | e_1, \ldots, e_N)$ in (6) is related to the version sampling distribution $P(\Pi = \pi)$ in (2) in any one of the following two ways: 1) they are identical, where the explicit dependence on the influences has been surpressed in $P(\Pi = \pi)$, or 2) one sampling distribution is obtained from the other by averaging over the possible values of the influences. That is, $P(\Pi = \pi) = \mathbb{E}\left[P\left(\Pi = \pi | E_1, \ldots, E_N\right)\right]$.

So, scenarios with quite different common influences, even affecting different phases of development, can be reduced to a common mathematical form from the viewpoint of conditional independence relations. We can thus formulate theorems that depend only on the presence of common influences.

In the following sections we discuss the implications of these models. We state the mathematical results, while the details, derivations and proofs can be found in the appendices.

C. *Effect of common influences on expected system* pfd

Consider a system development process as in Fig. 4, with multiple, mutually independent common influences $E_1, ...E_N$. There are also influences (I_A, J_A and I_B, J_B) that affect the development of only one version; we remove their nodes from the BBN to obtain Fig. 6 (by "marginalizing" as in section IV-B). Then, in Fig. 6 the only ancestor nodes of the Π_A and Π_B nodes are $E_1, ...E_N$, that model influences common to the two version development processes. The version developments are not independent; the ISA does not hold. The events "Π_A fails" and "Π_B fails", are *conditionally* independent, conditionally on the values of the common influences $E_1, ...E_N$ and the demand X:

$$\begin{aligned}\theta_{AB;e_1,\ldots,e_N}(x) :=\ & P\left(\omega\left(\Pi_A, x\right)\omega\left(\Pi_B, x\right) = 1 \middle| E_1 = e_1, \ldots, E_N = e_N\right) \\ =\ & \theta_{A;e_1,\ldots,e_N}(x)\ \theta_{B;e_1,\ldots,e_N}(x)\end{aligned} \tag{7}$$

where the three "θ" terms denote the difficulty functions for the system, for Π_A and for Π_B, given a demand, x, and values for all of the common influences.

As a consequence, the system difficulty function is (see Appendices B and C),

$$\theta_{AB}(x) = \theta_A(x)\theta_B(x) + \operatorname{Cov}\left[\theta_{A;E_1,\ldots,E_N}(x), \theta_{B;E_1,\ldots,E_N}(x)\right] \tag{8}$$

Differently from the EL and LM models, failures of the two versions are no longer independent conditionally on each demand: $\theta_{AB}(x) \neq \theta_A(x)\theta_B(x)$. The common influences invalidate the ISA and thus the form of conditional independence of failures that underpins the EL and LM models.

To illustrate, consider this thought experiment. Suppose that we know the difficulty of a certain demand x for version A, $\theta_A(x)$, and that the covariance term in (8) is positive on demand x; and that we observe that version A in fact fails on x. We should then consider likely that during development the values of the common influences $E_1, \ldots, E_N$ happened to take values $e_1, \ldots, e_N$ such that demand x was more "difficult" – more likely to become a failure point – than on average: $\theta_{A;e_1,\ldots,e_N}(x) \geq \theta_A(x)$. Recall that $\theta_A(x)$ is an average of $\theta_{A;e_1,\ldots,e_N}(x)$ over all possible values of the influences (all possible instances of the development process); the evidence of version failure shows that the process as it actually happened this time was probably more likely to make x a failure point. But both developments were exposed to these values $e_1, \ldots, e_N$ of the common influences; the positive covariance term then tells us that demand x was also especially difficult "this time" for the other version: $\theta_{B;e_1,\ldots,e_N}(x) \geq \theta_B(x)$.

From equation (8), the expected probability of common failure (i.e. expected *pfd* of the 1-out-of-2 system) is obtained by averaging $\theta_{AB}(x)$ over all possible demands, yielding:

$$\mathbb{E}\left[\theta_{AB}(X)\right] = \mathbb{E}\left[pfd_A\right]\ \mathbb{E}\left[pfd_B\right]\ +\ \operatorname{Cov}\left[\theta_A(X), \theta_B(X)\right]\ + \mathbb{E}\left[\ \operatorname{Cov}\left[\theta_{A;E_1,\ldots,E_N}(X), \theta_{B;E_1,\ldots,E_N}(X)\ \middle|\ X\right]\ \right] \tag{9}$$

Note that if there are no common influences, all covariance terms with respect to influences become zero, and this equation simplifies to equation (5), the LM model.

We have shown that common influences affect the expected system *pfd*. Given the scenario in Fig.4, if one could remove all the common influences *without changing the marginal sampling distributions of the two versions*, the expected system *pfd* would change. If each conditional covariance in the final term of (9) were positive, removing the common influences would be an improvement, perhaps a large one: isolating the version development processes would be beneficial, as commonly believed. Such a mathematically clean change might be infeasible in real life; but we can use this style of reasoning to identify categories of changes that would be beneficial.

---

[9] In Fig. 6, all common influences are mutually independent, a property that we will exploit shortly. If Fig. 4 had sets of non-mutually independent common influences, then to achieve the canonical form in Fig. 6 we would first merge each such set into a single r.v. (a "vector" r.v., whose set of possible values is the Cartesian product of the sets of possible values of all the r.v.s thus "merged"). The most general case of Fig. 6 would have a single common parent for the nodes Π_A, Π_B, as in Fig. 7, representing "all common influences affecting the two developments".

V. PREFERENCES BETWEEN DEVELOPMENT POLICIES

In this section we study scenarios in which our models imply a ranking between alternative policies for the development of a diverse-redundant system: the "better" policy guarantees *better or equal* mean system *pfd* than the alternative policy. We identify *sufficient conditions* for such rankings. We seek scenarios in which the models would actually help in decision making: sufficient conditions that one may recognise as approximately satisfied in the real world situation in which one is called to choose between possible policies.

A. Mirrored version development processes

An interesting special case is that of two version development processes being (stochastically) *mirrored*, by which we mean that the BBN parts that describe the developments of channel A and channel B (e.g. in Fig. 5 and Fig. 6) have identical nodes and edges and the same distributions for all their random variables: the two processes are probabilistically "identical", as in the EL model but with the added possibility of common influences. It may then be possible to *decouple* the version development processes with respect to a common influence, i.e., replace the common influence with two influences, one for each version, both having the same probability distribution as the common influence they replace, *without changing the rest of the version development processes*.

An example is the replacement of a process for randomly generating test cases, each test case then being used on both versions, with two stochastically identical, but independent, test case generation processes, one for each version [21]. The test cases are only used for testing, and testing results are only a function of the test cases and of the code under test, so no conditional probability distributions in the BBN change when we "decouple". Another example is choosing one compiler (from a range of suitable ones) for generating object code for both channels, *vs.* choosing two compilers, according to the same distribution, independently. The compiler affects which object code is subjected to linking, testing, etc, but the outputs of these further steps depend only on the input submitted to them, not on how it was generated. So, this decoupling does not change any conditional probability in the BBN.

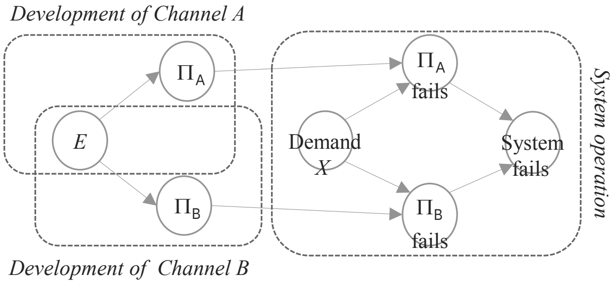

Fig. 7. BBN for a system with one common influence between the development processes. For example, if E is the set of test cases chosen, the same test cases are used to test both versions.

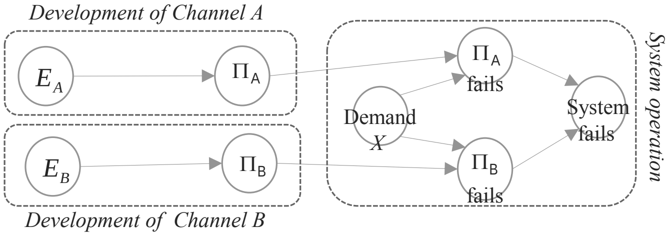

Fig. 8. The system development process of Fig. 7, after "decoupling" with respect to influence E. If E was a selection of test cases according to some method, here the two teams choose their test cases *independently*, but via *the same method* used in Fig. 7. The version sampling distribution and difficulty function for each channel are identical to Fig. 7 .

"Decoupling" transforms a graph like that in Fig. 7 into another graph, as in Fig. 8, where E_A and E_B have identical distributions to that of E in Fig. 7. To see how "decoupling" affects system reliability, we derive the mathematical form of the expected system *pfd* before and after decoupling. Both are given by averaging the system difficulty function $\theta_{AB}(x)$ over all possible demands, but, we will see, the two system difficulty functions differ in a way that implies a preference.

Consider a mirrored development process with N mutually independent common influences $E_1, \ldots, E_N$. Before decoupling, the system difficulty function $\theta_{AB}(x)$ has the form given by (8), since this equation results from (7). So, on each demand x, the system difficulty function is simply a weighted sum over the values of $\theta_{AB;e_1,\ldots,e_N}(x)$, as shown in Appendix B:

$$\begin{aligned}\theta_{AB}(x) &= \sum_{e_N}\left[\ldots\left[\sum_{e_1}\left[\theta_{AB;e_1,\ldots,e_N}(x)\right]P(E_1=e_1)\right]\ldots\right]P(E_N=e_N)\\ &= \mathbb{E}\left[\mathbb{E}\left[\ldots\mathbb{E}\left[\theta_{AB;E_1,\ldots,E_N}(x)\middle|E_2,\ldots,E_N\right]\ldots\middle|E_N\right]\right]\end{aligned} \tag{10}$$

Without loss of generality[10] and using the fact that the two version development processes are identically distributed[11], we rewrite the innermost sum in (10) as:

$$\begin{aligned}\sum_{e_1}\left[\,\theta_{AB;e_1,\ldots,e_N}(x)\,\right]P\left(E_1=e_1\right) &= \mathbb{E}[\,\theta_{AB;E_1,e_2,\ldots,e_N}(x)\,]\\ &= \mathbb{E}[\,\left(\theta_{A;E_1,e_2,\ldots,e_N}(x)\right)^2\,]\\ &= \left(\mathbb{E}[\,\theta_{A;E_1,e_2,\ldots,e_N}(x)\,]\right)^2 + \mathrm{Var}\left[\,\theta_{A;E_1,e_2\ldots e_N}(x)\,\right],\end{aligned} \tag{11}$$

where $\mathrm{Var}_{E_1}\left[\theta_{A;E_1,e_2\ldots e_N}(x)\right] \geq 0$, since it is a variance.

Upon decoupling, the single common influence E_1 is replaced by two independent, identically distributed[12] influences – E_A and E_B say – one for each version development process. Thus, conditional on the realisations e_A, e_B, $e_2,\ldots,e_N$ of the influences, the conditional system difficulty function factors into a product of conditional difficulty functions, one for each version development process (*cf.* (7)).

$$\begin{aligned}\theta_{AB;e_A,e_B,e_2\ldots,e_N}(x) &:= P\left(\omega\left(\Pi_A,x\right)\omega\left(\Pi_B,x\right)=1\;\middle|\;E_A=e_A,\;E_B=e_B,E_2=e_2,\ldots,E_N=e_N\right)\\ &= \theta_{A;e_A,e_2,\ldots,e_N}(x)\,\theta_{B;e_B,e_2,\ldots,e_N}(x)\end{aligned} \tag{12}$$

Thus, the system difficulty associated with an arbitrary demand x is given by the following expectation.

$$\begin{aligned}\theta_{AB}(x) &= \sum_{e_N}\left[\ldots\left[\sum_{e_2}\left[\sum_{e_B}\sum_{e_A}\left[\theta_{AB;e_A,e_B,e_2\ldots,e_N}(x)\right]P(E_A=e_A)P(E_B=e_B)\right]P(E_2=e_2)\right]\ldots\right]P(E_N=e_N)\\ &= \mathbb{E}\left[\mathbb{E}\left[\ldots\mathbb{E}\left[\theta_{AB;E_A,E_B,E_2\ldots,E_N}(x)\middle|E_2\ldots,E_N\right]\ldots\middle|E_N\right]\right]\end{aligned} \tag{13}$$

Comparing (10) and (13), we note that the difference in the weighted sums before and after decoupling lies in the innermost sum: expectation with respect to E_1 has been replaced with an expectation with respect to the pair E_A and E_B. Expanding the innermost sums in (13) as follows shows the effect of this difference.

$$\begin{aligned}\sum_{e_B}\sum_{e_A}\left[\theta_{AB;e_A,e_B,e_2\ldots,e_N}(x)\right]P\left(E_A=e_A\right)P\left(E_B=e_B\right) &= \mathbb{E}[\;\theta_{AB;E_A,E_B,e_2,\ldots,e_N}(x)\;]\\ &= \mathbb{E}[\;\theta_{A;E_A,e_2,\ldots,e_N}(x)\;\theta_{B;E_B,e_2,\ldots,e_N}(x)\;]\\ &= \mathbb{E}\left[\;\theta_{A;E_A,e_2,\ldots,e_N}(x)\;\right]\;\;\mathbb{E}\left[\;\theta_{B;E_B,e_2,\ldots,e_N}(x)\;\right]\\ &= \left(\mathbb{E}[\,\theta_{A;E_1,e_2,\ldots,e_N}(x)\,]\right)^2\end{aligned} \tag{14}$$

Equations (11) *and* (14) *are identical (that is, the terms are the same), except that decoupling has eliminated the non-negative variance term in* (11); this holds for each possible realisation of the circumstances of development $e_2,\ldots,e_N$ and each demand x. In summary, the common influence E_1 creates variance terms, which would be added up through all the sums in (10), that "decoupling" with respect to E_1 removes: the expected system *pfd* from (10) is never smaller than that from (13); this justifies the following preference criterion.

> **Preference criterion 1: Decoupling of mirrored version development processes**. Given two mirrored development processes with a finite number of independent, common influences, substituting one of the common influences, say E, with two independent influences (one for each process), with the same distribution as E, yields better (or at worst unchanged) expected system *pfd*.

Note that:

- after applying this "decoupling", the resulting processes are still mirrored and, therefore, this criterion still applies: "decoupling" with respect to *any finite number* of common influences is an improvement;

---

[10] These equations are invariant under permutations of the common influences since: 1) there is a finite set of mutually independent common influences, and 2) the difficulty functions are non-negative. The mutual independence of the common influences is best seen in Fig. 6, which the equations describe. In the BBN there is symmetry among the common influences. Creating probabilistic dependence between two common influences, e.g. as an arc joining them, would destroy the symmetry. See also footnote 9.

[11] So that $\theta_{A;e_1,e_2,\ldots,e_N}(x) = \theta_{B;e_1,e_2,\ldots,e_N}(x)$ for all $e_1,\ldots,e_N,x$.

[12] That is, E_1, E_A and E_B have the same probability distribution.

- just *removing* a common influence that affects both development processes does not guarantee improvement; leaving all the other probabilities unchanged is also necessary. E.g., one could remove a common influence “common selection of test cases” by just omitting that testing stage altogether. But this would change the two version sampling distributions, presumably for the worse, and thus may cause *worse* expected system *pfd*. The beneficial “decoupling” defined here, instead, does not change the two version sampling distributions, but only their joint distribution, and this turns out to be an improvement.

B. Non-mirrored version development processes

For development processes that are *not* mirrored – that produce program versions according to different version sampling distributions – what are the effects of decoupling, if any?

The version development processes differ: there exist values of $e_1, \ldots, e_N, x$ for which $\theta_{A;e_1,e_2,\ldots,e_N}(x) \neq \theta_{B;e_1,e_2,\ldots,e_N}(x)$. Instead of (11), we have

$$\begin{aligned}\mathbb{E}\left[\ \theta_{AB;E_1,e_2,\ldots,e_N}(x)\ \right] &= \mathbb{E}\left[\ \theta_{A;E_1,e_2,\ldots,e_N}(x)\ \theta_{B;E_1,e_2,\ldots,e_N}(x)\ \right]\\ &= \mathbb{E}\left[\ \theta_{A;E_1,e_2,\ldots,e_N}(x)\ \right]\mathbb{E}\left[\ \theta_{B;E_1,e_2,\ldots,e_N}(x)\ \right] \quad + \mathrm{Cov}\left[\ \theta_{A;E_1,e_2,\ldots,e_N}(x),\ \theta_{B;E_1,e_2,\ldots,e_N}(x)\ \right]\end{aligned} \tag{15}$$

with a covariance term (with respect to E_1) where (11) has a variance. Covariances may be negative, positive or zero: decoupling with respect to this influence, by eliminating the covariance term, might increase, reduce, or leave unchanged, the expected system *pfd*. Decoupling is beneficial if the covariance term in (15) is positive (for all values of $E_2, \ldots, E_N, x$).

> **Preference criterion 2: Decoupling of diverse version development processes**. Given two development processes with a finite number of independent, common influences, if the two processes’ difficulty functions have non-negative covariance with respect to some common influence E, for all possible values of all of the other common influences and of the demand, then substituting E with two independent influences (one for each process) with the same distribution as E, yields better (or at worst unchanged) expected system *pfd*.

To use this result in concrete decisions, one needs first to be able to trust the positive covariance assumption, without estimating the θs and expected *pfd*s in the equations (which is normally impossible). Scenarios for which this “positive covariance” expectation is reasonable do exist. For instance, if both versions are purpose-built for the same system, and the time or effort available for developing each version may vary due to some factor that affects both – say, the client changing the delivery deadline – then one would expect that increasing the time available will reduce “difficulty” for every demand, irrespective of other influences, while reducing the time will increase “difficulty”: the two difficulty functions are, for each demand, monotonically decreasing functions of the common influence, and thus create a positive covariance term. [13]

Given such a scenario, Criterion 2 implies that for such custom-built versions, if there is an option to “decouple” (with respect to common influences that have such correlated effects on both version developments) *without otherwise changing the statistics of the two version development processes*, this option should be taken. Likewise, given a choice between pairing off-the-shelf versions that were originally affected by such a common influence (with unknown value) and others that were not – the development processes being otherwise stochastically equivalent in the two scenarios – we should expect better system *pfd* from the latter. The “keeping everything else equal” requirement is, of course, the main challenge. [14]

A dual theorem also applies: if the conditions of preference criterion 2 are satisfied, except that the covariance term with respect to E is negative, then *not decoupling* with respect to E will *improve* the expected system *pfd*. This suggests the appealing idea of introducing random influences such that whenever on some demands they happen to hinder one version development team, they also help the other team. This possibility was first discussed with respect to the choice of test cases [21], noting that it was hard to imagine strategies giving such negative covariance. We describe an example scenario, based on project management dynamically ensuring diversity of some aspect of design, in Appendix E.

C. Homogeneous versus diverse development processes

Finally, we return to the case of two mirrored version development processes with mutually independent common influences. A plausible way of “increasing diversity” is to require one of the two development teams to change its own process in some aspect (and thus change the probability distributions associated to some node in the BBN), with the constraint that this alternative process should be different from, but no worse than, the original process. What changes might be the technology used, or the testing methods, or the algorithms implemented. Let us call the two processes α and β.

---

[13] Non-negative covariance is guaranteed if the two difficulty functions $\theta_{A;e_1,e_2,\ldots,e_N}(x)$ and $\theta_{B;e_1,e_2,\ldots,e_N}(x)$ are monotonic functions (both non-increasing or both non-decreasing) of E_1, irrespective of the values $e_2, \ldots, e_N, x$ (*cf* Appendix D).

[14] To illustrate this challenge, consider e.g. the case of the common influence “time available from project start to delivery”. A project manager could try to “decouple” with respect to this influence by e.g. buying one of the two versions, say B, “off the shelf”: its development would certainly not have been subjected to the same fortuitous variations in development time as purpose-built version A; but would it be close enough to being stochastically identical to the development of a hypothetical purpose-built version B, so that one can trust the off-the-shelf option for B to be advisable per Criterion 2? Or the project manager could perhaps partition the demand space into two subsets $\mathcal{X}_\mathcal{A}$ and $\mathcal{X}_\mathcal{B}$, and mandate how much time team A must devote to demands in $\mathcal{X}_\mathcal{A}$ and team B to those in $\mathcal{X}_\mathcal{B}$. Thus, for demands in – for instance – $\mathcal{X}_\mathcal{A}$, the difficulty function $\theta_A(x)$ is made independent of the influence “time available for system testing”, while $\theta_B(x)$ depends on it: $cov(\theta_A(x), \theta_B(x))$ with respect to this influence is 0. But, even assuming that this management intervention makes concrete sense in this project, it makes difficulty functions $\theta_A(x)$ and $\theta_B(x)$ different from what they would be without it: again, Criterion 2 would not automatically apply. Being aware of a commonality for which “decoupling” might be beneficial does not imply that such “decoupling” is actually feasible.

Given *no* common influences, Littlewood and Miller [12] proved that this change is an improvement – the expected system *pfd* of a system in which one version is produced by α and the other by β is no worse than that of a system in which both versions are produced by the same process – if α and β are equivalent in that they satisfy an "indifference" condition: a system made of two versions, both produced by α, has the same expected *pfd* as a system of two versions produced by β. If I have no reason for believing that either process, used for both versions, would produce on average a superior two-version system, the rational decision is to apply different processes to the two versions.

We prove in Appendix F that this inequality also holds if the two version development processes are not independent but have common influences, and thus:

> **Preference criterion 3: Diversification between version developments**. If using either a process α exclusively, or a process β exclusively to develop two versions for a 1-out-of-2 system gives the same expected system *pfd*, then using process α for one version and process β for the other yields better (or at worst unchanged) expected system *pfd*.

Acting upon this "criterion for improvement" is, in a sense, irreversible. Once we alter two mirrored version development processes by "diversifying" some part of them, and thus turning variance terms in our expression for the average system *pfd* into covariance terms, adding a common influence will not reintroduce variance terms. That is, adding some "coupling" may make the system less reliable on average, but will not produce in the equations the feature that is a sufficient condition for this decreased reliability.

D. *Influences of a development team on another*

Suppose a team is allowed to send suggestions or questions to the other team; or to the project manager, who then issues instructions to both teams. With reference to Fig. 5, this could mean e.g. an extra node, to represent the contents of the communication, with parents in the "Channel A's development" part, and edges leading to, e.g., the "Specification clarification" node (if team A prompts a clarification) or directly to nodes in the "Channel B's development" part (if team A talks directly to team B). One could similarly allow team B to communicate to team A and/or the management. Such edges from the "Channel development" parts of the BBN to the rest complicate the topology that we have assumed. The effects on the preference criteria depend on what questions we ask. For instance:

- to decide whether to "decouple" with respect to the "System test demands" common influence, we can just merge (footnote 9) all the nodes to the left of "versions before system testing" into a single common influence. The resulting BBN belongs to the types already studied, and one can check for applicability of criterion 1 or 2;
- to decide, instead, whether these communication paths between the teams are an improvement or not, we have as yet no topology-based preference criterion. The answer depends on the values of the conditional distributions, which determine whether these communications would, on average, improve the system development process, e.g. whether the potential benefit from removing specification errors (improving system reliability) outweighs the potential harm of sharing views that foster common errors by both teams.

VI. DISCUSSION

We have studied questions about the effects of project management, or procurement, policies on the system *pfd* of 1–out–of–N systems. Between two multiple-version development processes that appear sound, which one should be picked? Or, given a tried and tested process will a certain proposed change to it be an improvement? Since the system development process does not fully determine the resulting system *pfd*, we need to reason in terms of probabilities and probability distributions. There is normally not enough statistical evidence to suggest such probabilities; yet, rigorous probabilistic reasoning at least clarifies whether the assumptions that one believes to be true do support a specific decision. We have looked for pairs of alternative development processes for which mathematical modelling can show which alternative to prefer.

A. *Mathematical view of results*

Until now, most interesting results derived from the seminal work by Eckhardt and Lee and Littlewood and Miller seemed only to apply to the narrow case of independently developed versions with identical expected *pfd*. We have removed this limitation via a natural, general and powerful extension of these models. Via the abstraction of "common influences" – random variables affecting version development – many scenarios of system development, with various factors affecting system failure, can be represented and reduced to a common representation as in Fig. 6 and Section IV-C.

"Common influences" are not limited to development. For instance, to take into account physical failures as well as software-caused failures, our BBNs can represent common stress factors like ambient temperature or common shocks as extra parent nodes of the nodes "Π_A fails" and "Π_B fails".

Our models are quite abstract, in that they refer to functions that cannot usually be estimated in practice, but they:

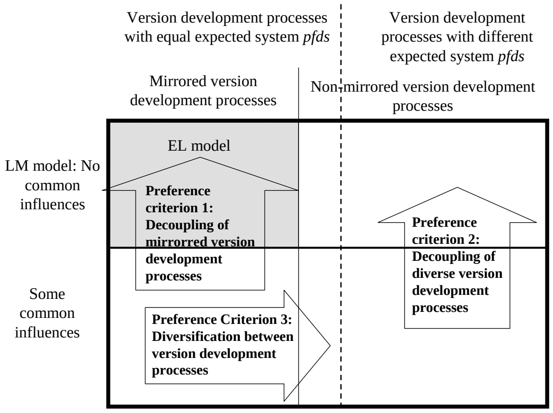
Version development processes with equal expected system *pfds*
Version development processes with different expected system *pfds*
Mirrored version development processes
Non-mirrored version development processes
EL model
LM model: No common influences
Some common influences
Preference criterion 1: Decoupling of mirrorred version development processes
Preference criterion 2: Decoupling of diverse version development processes
Preference Criterion 3: Diversification between version development processes

Fig. 9. The space of possible assumptions about a system development process, and its subsets on which the various results apply. The arrows show "preference criteria": following an arrow between subsets of scenarios can only change expected system *pfd* for the better. The arrow on the right stands for preference criterion 2 when the common influence used for "decoupling" the processes induces positive covariance between difficulty functions; with negative covariance, the arrow would point downward.

- clarify the relationship between the intuitive ideas of "separation", " independence", "diversity of process", the formal concepts of independence and correlation, and measures of interest such as the *pfd*;
- offer some direct, practical help for decision making, like our three "preference criteria" among processes for developing two-version systems, based on sufficient conditions which we think people will recognise to match their assumptions in certain practical decision problems, e.g. "indifference" between two development processes.

Previous results can now be seen as special cases. For instance, it was previously shown that with two isolated and stochastically equivalent version development processes, a common test suite between the two processes would increase the probability of common failure [21]. Our preference criteria generalise this conclusion: for *any finite* number of stages of testing, given equal (possibly non-independent) development processes for the pre-test versions, separate, *independent generation of the test suites for the two versions is preferred* to choosing the same suite for the two versions.

Fig. 9 shows how our results widen the range of scenarios in which mathematically founded preferences hold between alternate ways of running multiple-version development. Our "preference criteria" specify changes that improve the system development process by shifting it from one domain to another in Fig. 9. They improve over previous theory by giving sufficient conditions for answering questions such as, "When is combining multiple ways of 'forcing' diversity more beneficial than 'forcing' diversity in just one way?" ([5], [22]).

Representing the models as Bayesian Belief Networks seems beneficial in itself: conditional independence relationships are made explicit, and their implications can be seen by applying simple rules based on the graph's topology.

Our results also clarify the meaning of "independence" between version developments, and somewhat reduce its importance. Given commonalities between the developments of two versions, whether the two developments are stochastically independent depends on how much we assume to be uncertain about them. Independence holds, for all of the scenarios of real-world developments we have modelled, *once all the random factors that affect both versions* have taken specific values. This latter viewpoint is the better one for answering some interesting questions. For example, to study the effect of adding a common influence at a certain stage of the version development processes, it helps to assume as determined, and thus independent, all the *previous* stages. The added influence then violates this independence, and one can focus on whether its effects would be positive or negative. These observations are detailed further in appendix G.

B. Updates to accepted principles and opinions

A major question we addressed is whether the insights of the earlier "EL" and "LM" models [7] remain valid without the extreme assumption of strictly independent version development processes. The main message from the "EL" result was that a prudent (pessimistic) assessor, given identical version development processes, should assume that the mean probability of system failure is worse than the product of the mean probabilities of channel failure. This message *still holds*: given mirrored version development processes, the ISA was indeed the most optimistic assumption (*cf.* (11)), thus the EL message applies *a fortiori* if it is violated. Note that "forcing" diversity alone, without satisfying our preference criteria, does not even guarantee better expected system *pfd* than that implied by the EL assumptions.

As for guidance about how to produce diverse systems, our results *support*, for many scenarios, the common beliefs in strong "separation" and "diversification" between version development processes, but also show their limits. *Any random common influence* affects the expected system *pfd*, but this only matters in practice if, as in our preference criteria, we compare concrete alternatives that differ in these influences. And our models clearly describe cases that violate "common-sense" generalizations:

1) common influences may reduce diversity and yet improve system *pfd* (e.g., with additional testing on common test cases); 2) common influences may improve system *pfd* by *increasing* diversity ((8) and Appendix E)). This suggests dynamically altering the process applied to one version to make it “as different as possible” from that applied to the other, not just trying to force such differences via pre-development decisions.

Our preference criteria still depend on restrictive sufficient conditions. We believe that often in software development the decision makers can make informed judgements about whether these assumptions hold, even if only approximately. Then, they can make decisions consistent with these judgements by using our criteria. For example, preference criterion 2 (about “decoupling” with respect to a common influence) requires difficulty functions with “non-negative covariance . . . for all possible values of all of the other common influences and of the demand”: if one expected large non-negative covariance (consistent effects of a certain influence on both versions’ probability of failure) for *all the most likely* values of other influences and demands, he could justifiably expect that “decoupling” per Criterion 2 would be an improvement.

C. *Directions for further research*

Further study with this modelling approach should look for other “preference criteria”, with sufficient conditions that can be recognised in practical situations. Conditions for communications initiated by the version developments teams (section V-D) to be advantageous or damaging would be especially interesting; as would be additional examples of system development policies that create useful, negative covariance between the difficulty functions of the two processes.

1-out-of-2 systems are an important category of systems, and illustrate the basic problems in managing “diversity”, but results applicable to other fault-tolerant architectures would be desirable. Littlewood and Miller [12] showed that many results do not extend in intuitive ways from the 1-out-of-2 case, and further subtle, counter-intuitive effects are possible.

Even with such further developments, theorems, apart from improving understanding, can answer only few of the questions about how to develop diverse software (or any other human endeavour). Our results implicitly outline the large set of other questions that can only be answered empirically, and often only after the fact. First, when the sufficient conditions for our preference criteria are not satisfied, decisions must be based on experimental evidence about what works best, or, in its absence, on educated, subjective judgement. In addition, our criteria are just inequalities: they indicate the “better” one between two options, but not the *magnitude* of the advantage to be expected from it. Experimental work can measure examples of such magnitudes,to inform judgement. even though general rules for predicting the effects in a new development projects seem out of reach [23], [24].

All these results concern *expected values* of system *pfd*, useful for decisions about how to build a diverse system. For *assessing a delivered system*, instead, they can help to claim that the development approach was appropriate – a claim that has a major role in how critical software is currently assessed – but nothing more. Assessing the product actually delivered is obviously essential, though difficult for the extreme reliability levels often sought via software diversity [8], [9].

D. *Summary*

These results greatly extend the set of situations for which preferences between ways of developing diverse software can be stated on a purely mathematical basis. They support common beliefs about the importance of separation and diversity between development processes towards achieving failure diversity, but rigorously delimit the premises under which these beliefs are justified. Formal probabilistic modelling reveals non-intuitive results, like the conditions under which removing “commonalities” between version developments, or intentionally diversifying them, is certainly helpful (but may or may not be feasible); or is certainly damaging; or has uncertain effect.

ACKNOWLEDGMENTS

This work was partially supported by the U.K. Engineering and Physical Sciences Research Council (EPSRC) under projects DIRC and INDEED, and by the the DISPO project, funded under the C&I Nuclear Industry Forum (CINIF) Nuclear Research Programme by EDF Energy Limited, Nuclear Decommissioning Authority (Sellafield Ltd, Magnox Ltd), AWE plc, Urenco UK Ltd and Horizon Nuclear Power. The views expressed in this paper are those of the author(s) and do not necessarily represent the views of CINIF members. CINIF does not accept liability for any damage or loss incurred as a result of the information contained in this paper. We thank Peter Bishop and the anonymous TDSC reviewers for their comments on earlier versions of this paper.

REFERENCES

[1] L. Strigini, “Fault tolerance against design faults,” in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications,*, H. Diab and A. Zomaya, Eds. J. Wiley & Sons, 2005, pp. 213–241.
[2] L. Pullum, *Software Fault Tolerance Techniques and Implementation*, ser. Computer Security Series. Artech House, 2001.
[3] A. Avizienis, “The methodology of N-version programming,” in *Software Fault Tolerance*, M. Lyu, Ed. J. Wiley & Sons, 1995, pp. 23–46.
[4] M. Lyu and Y. He, “Improving the N-version programming process through the evolution of a design paradigm,” *IEEE Transactions on Reliability*, vol. R-42, pp. 179–189, 1993.

[5] B. Littlewood and L. Strigini, “A discussion of practices for enhancing diversity in software designs,” Centre for Software Reliability, City University, DISPO project technical report LS-DI-TR-04, 2000. [Online]. Available: http://openaccess.city.ac.uk/275/

[6] P. T. Popov and L. Strigini, “Conceptual models for the reliability of diverse systems - new results,” in *28th International Symposium on Fault-Tolerant Computing (FTCS-28)*. Munich, Germany: IEEE Computer Society Press, 1998, pp. 80–89.

[7] B. Littlewood, P. Popov, and L. Strigini, “Modelling software design diversity - a review,” *ACM Computing Surveys*, vol. 33, pp. 177–208, 2001.

[8] B. Littlewood and L. Strigini, “Validation of ultra-high dependability for software-based systems,” *Communications of the ACM*, vol. 36, pp. 69–80, 1993. [Online]. Available: http://openaccess.city.ac.uk/1251/

[9] R. W. Butler and G. B. Finelli, “The infeasibility of quantifying the reliability of life-critical real-time software,” *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 3–12, 1993.

[10] J. C. Knight and N. G. Leveson, “An experimental evaluation of the assumption of independence in multi-version programming,” *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 96–109, 1986.

[11] D. E. Eckhardt and L. D. Lee, “A theoretical basis for the analysis of multiversion software subject to coincident errors,” *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1511–1517, 1985.

[12] B. Littlewood and D. R. Miller, “Conceptual modelling of coincident failures in multi-version software,” *IEEE Transactions on Software Engineering*, vol. SE-15, pp. 1596–1614, 1989.

[13] B. Littlewood, P. Popov, L. Strigini, and N. Shryane, “Modelling the effects of combining diverse software fault removal techniques,” *IEEE Transactions on Software Engineering*, vol. SE-26, pp. 1157–1167, 2000.

[14] B. Littlewood and L. Strigini, “Redundancy and diversity in security,” in *ESORICS 2004, 9th European Symposium on Research in Computer Security*, ser. LNCS, P. Ryan and P. Samarati, Eds. Sophia Antipolis, France: Springer-Verlag, 2004, pp. 423–438.

[15] L. Strigini, A. A. Povyakalo, and E. Alberdi, “Human-machine diversity in the use of computerised advisory systems: a case study,” in *DSN 2003, International Conference on Dependable Systems and Networks*, San Francisco, U.S.A., 2003, pp. 249–258.

[16] Y. C. B. Yeh, “Design considerations in Boeing 777 fly-by-wire computers,” in *3rd High-Assurance Systems Engineering Symposium (HASE)*. Washington, DC, USA: IEEE Computer Society Press, 1998, pp. 64–73.

[17] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. Kelly, “An experimental evaluation of software redundancy as a strategy for improving reliability,” *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692–702, 1991.

[18] S. Lauritzen, *Graphical Models*, ser. Oxford Statistical Science Series. Clarendon Press, Oxford, 1996, vol. 17.

[19] CACM, “Real-world applications of Bayesian networks,” *Communication of the ACM, Special Issue*, vol. 38, no. 3, 1995, – SHIP T046.

[20] P.-J. Courtois, B. Littlewood, L. Strigini, D. Wright, N. Fenton, and M. Neil, “Bayesian Belief Networks for safety assessment of computer-based systems,” in *System Performance Evaluation: Methodologies and Applications*, E. Gelenbe, Ed. CRC Press, 2000, pp. 349–363.

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

[22] P. Popov, L. Strigini, and A. Romanovsky, “Choosing effective methods for design diversity - how to progress from intuition to science,” in *SAFECOMP ’99, 18th International Conference on Computer Safety, Reliability and Security*, ser. LNCS, M. Felici, K. Kanoun, and A. Pasquini, Eds. Toulouse, France: Springer, 1999, pp. 272–285.

[23] X. Cai, M. R. Lyu, and M. A. Vouk, “An experimental evaluation on reliability features of n-version programming,” in *16th International Symposium on Software Reliability Engineering (ISSRE 2005)*. IEEE Computer Society Press, 2005, pp. 161–170.

[24] P. Popov, V. Stankovic, and L. Strigini, “An empirical study of the effectiveness of ’forcing diversity’ based on a large population of diverse programs,” in *23rd International Symposium on Software Reliability Engineering (ISSRE 2012)*. IEEE Computer Society Press, 2012.