Title of Document:	# A discussion of practices for enhancing diversity in software designs
Title of Project:	DIverse Software PrOject (DISPO) (British Energy - ex Scottish Nuclear - Project under Contract PP/96523/MB)
Authors:	Bev Littlewood, Lorenzo Strigini
Document Identifier:	DISPO LS_DI_TR-04_v1_1d
Date Last Modified	23 November 2000
Version:	1.1d
Document Type:	DISPO report (Deliverable)
Intended Recipients:	Public distribution
Confidentiality:	Cleared for publication
Contact	Lorenzo Strigini <strigini@csr.city.ac.uk>
Document Location	Lorenzo' Strigini's Macintosh

A discussion of practices for enhancing diversity in software designs

Bev Littlewood, Lorenzo Strigini
Centre for Software Reliability, City University
Northampton Square, London EC1V OHB, UK
{b.littlewood,l.strigini@csr.city.ac.uk}

DISPO Project Draft Technical Report LS_DI_TR-04,
Version 1.1d, 23 November 2000

Abstract

This report discusses the practices which have been used or recommended for increasing the degree of diversity between redundant implementations of software or software-based systems. Its purpose is to give useful indications for designers, project managers and safety/reliability assessors in deciding about how great an advantage should be expected from the use of these practices, in absolute and in comparative terms. Existing knowledge does not allow one to state any strong general recommendations, but it is possible to improve on the intuitive justifications usually given for these various practices. This report clarifies the ways the various practices are conjectured to aid system reliability, the factors that should affect their efficacy, and thus, for a practitioner, the aspects of a specific project situation that need to be considered to inform decisions.

Thus this report is meant to improve on the many recommendations available in the literature by a more rigorous analysis of the support available for individual recommendations and for decision between them, on the basis of existing known evidence about diversity, of general experience in software engineering and of the result of our reliability modelling work.

An executive summary gives the highlights of the report and a guide to the topics treated. The other sections are an introduction giving the scope and background of the work; a general analysis of the factors affecting the achievement of useful diversity and the criteria for choosing among possible "diversity-seeking decisions" to this end, and a summary table of the considerations applying to each category of "diversity-seeking decisions", with explanations of detail in an appendix.

This report updates and supersedes the previous DISPO project report AT_DI-D-01-v1.7, "A list of intuitive diversity enhancing measures/practices", 20 February 1998, which was produced at the beginning of the DISPO project, to reflect our changed understanding at the end of the project. Parts of the old report have been eliminated as the corresponding topics are now covered by separate DISPO documents.

Contents

Executive summary

Design diversity is an attractive and popular defence against common-mode failures due to design faults in redundant systems. It is considered especially important when all channels in a critical system, e.g. a protection system, have to be realised as programmable systems. However, the abundant advice available about how to pursue diversity so as to best reduce common-mode failures is confusing and usually lacking convincing bases.

In this section, we indicate the topics studied in the various parts of this document, and summarise the conclusions reached.

Background and scope (Section 1)

Section 1 sets the background for this study: its role within the DISPO project, its scope, the common understanding of the problems in the technical community, the essential models and terminology necessary for the following discussion. This study was motivated by the issues raised by the possibility of completely software-based protection systems. Its focus was thus on systems with two diverse channels (versions), Boolean outputs and 1-out-of-2 redundancy, but many of its considerations apply beyond this area. Section 1.1 defines the scope of this study and outlines how other kinds of architectures using diversity may differ from these systems. In Section 1.2, we recall the common approaches followed in pursuing design diversity and define the term "diversity-seeking decision" (DSD): a decision available to system designers to attempt to promote failure diversity between two program versions. Common DSDs are, for instance, the decision that two protection channels will be developed by separate teams, and giving these teams specifications formulated in different notatons; making them use different programming languages; and using different microprocessors in the two channels. We defined the special term "DSD" to avoid the common confusion between descriptions of *decisions* made and descriptions of their desired or actual effects - the actual diversity achieved between versions or their failure behaviours. Decisions are obviously under our control, but we do not usually know their likely effects. Their actual effects can to some extent be measured after the fact, but are difficult to predict at decision time. A DSD is a choice of *differences* between two development processes. In section 1.3 we list the essential different meanings of "diversity" that have to be kept separated to avoid errors in analysing them, and in section 1.4 we recall the mathematical models (the "EL" and "LM" models) which are at the origin of our current understanding of "failure diversity" between the versions, the actual goal of interest.

General principles of analysis, and limits to the analysis (Sections 2.1 -2.5)

So, the topic of this report is how to choose DSDs for best effect on system reliability. This cannot be based on statistical evidence of their effectiveness, which is lacking because:

- there are many possible variations and combinations of DSDs;
- experiments (producing many different program versions under controlled conditions, and comparing their failure behaviours) could show, at great expense, how effective a DSD was in a particular case, but not whether that effectiveness would be retained when developing a different system;
- we should expect the relative effectiveness of DSDs to change between different project contexts: product type, company culture, dependability requirements, project constraints.

Instead of seeking general measures to attach to each DSD in the abstract, we need to understand the factors that make a DSD useful, and the effects of project conditions on the relative weights of these factors. To this end we can use some of the known experimental results, together with our general understanding of software development and software reliability. We can identify several mechanisms that explain the positive effects of DSDs on failure diversity, and are common to many DSDs, although active in different measures. Section 2 describes these mechanisms, the difficulties in ascertaining their role and power, and the general practical approaches available for choosing among DSDs.

First (sections 2.1, 2.2, 2.3) we describe various ways in which DSDs may promote failure diversity. These include:

- the "obvious" chain of cause and effects: a DSD directly ensures that developers are likely, if they make mistakes, to make *different* mistakes, which will cause any faults in the developed versions to be conceptually different, which will cause failures not to occur on the same demands for both versions; but also
- several other mechanisms: data diversity (intentional or accidental), such that even similar faults are only triggered in one of the channels; accidental differences in the effects of similar development mistakes; the fact that the constraints we create for development teams may indirectly create further diversity between the individual tasks in which they engage.

Any DSD should be based on some understanding of human error mechanisms. Section 2.4 is dedicated to this. Although psychology has delivered interesting insights into human error processes, important uncertainties remain when we consider the complex web of activities that forms software development, many of these activities being specifically aimed at detecting and removing the effects of previous errors. Two fundamental characteristics of human errors are that they occur in random fashion, but on the other hand they are affected by the characteristics of the tasks to be performed. The latter aspect would encourage us to select DSDs so as to diversify the tasks presented to the two development teams. The former aspect, together with the complexity of how development errors produce system failures, would indicate that the most important DSD is the most basic one, i.e., simple separation between development efforts.

Ideally, the correct way of choosing a DSD is to verify that the two development processes it creates are in a sense complementary: the demands on which the programs created by one process are more likely to fail must be those on which programs created by the other are less likely to fail. In addition, we normally wish each process on its own to be as good as practically possible. In practice, many difficulties stand in the way of such ideal choices:

- we seldom know in detail the strengths and weaknesses of the different methods that comprise our development processes, and what we do know is usually in terms of the likelihood of people making [certain kinds of] mistakes when using those methods, not of the likelihood that the defects caused by these mistakes cause failures on the same sets of demands, or the probability of such common failures in operation;
- system development includes many different, interacting activities. We often have ideas about the influence of a DSD on one activity, but not about its side-effects on other activities and the relative importance of all these influences. So, we have both many possible combinations of available choices for the various activities in a development process, and little support for comparing these combinations;
- there are trade-offs between increasing the reliabilities of the versions and their diversity. There are even cases, in theory, in which we should produce two versions with two processes that are much less than ideal rather than with better processes, because the increased diversity produces in the end higher *system* reliability. We are unlikely in practice to be able to recognise such a situation, but a dedicated subsection (2.7) deals with related questions of practical importance - when is it that applying a DSD to increase diversity may at the same time decrease version reliability? - and the available decision criteria;
- practical constraints (budget, staff, contractual obligations and so on) will normally reduce the number of possible choices, but often not the complexity of the factors to be compared.

We explain these difficulties in section 2.5, which is mostly a set of negative conclusions and warnings about the limits to the extent to which the efficacy of DSDs can be judged.

Criteria for choosing diversity-seeking decisions (Sections 2.6-2.8)

In section 2.6 we start outlining decision criteria that appear reasonable given the limited knowledge available. In general, the aim is to ensure the *possibility* of useful diversity, even though one cannot forecast whether this potential gain in actually achieved diversity will justify the cost. The basic step is, again, separation between development efforts. Any further DSD is probably desirable, but there is uncertainty because there are many steps between the decision (e.g., to use two different formalisms for specification) and the desired effect: a decreased probability of versions failure. For many of these steps, there is no indication that the supposed beneficial effect of the decision will propagate through them. So, there is often the theoretical

possibility that a decision intended to increase diversity will in practice decrease it, but it is reasonable to assume this not to be the case, so long as we have no evidence of it actually occurring.

Since the main concern is that excessive similarities between the problems posed to the two development teams (i.e., similarities in the "difficulties" they present - *cf* section 1.4) cause too high a risk of common failures, any DSD for which we can reasonably expect some degree of effectiveness appears desirable. The limits on adopting DSDs are cost constraints, organisational problems, or the fear of direct negative effects on system reliability. We will naturally adopt all the DSDs that have low cost and no evident negative side effects and appear potentially useful. For those among which we must instead choose, we have two main, possibly competing criteria: diversifying with higher priority those aspects of the development processes from which we expect the more substantial risk of common failures, despite the other precautions that we can adopt; and diversifying the processes "as completely as possible", to compensate for our limited ability to identify the main residual threats in high-quality processes.

From our analysis of DSDs, three essential groupings emerge based on the threats they primarily address: defences against high-level mistakes in defining system structure or algorithms, defences against coding (or generally, detail implementation) mistakes in implementing these high-level decisions, and defences against defects in the "support platform" (compilers-linkers-loaders, hardware and run-time executive or operating system).

The benefits of DSDs generally propagate (unidirectionally) across these three levels: e.g., imposing different high-level architectures for two versions, for the purpose of avoiding common mistakes in defining the basic algorithms, is likely also to cause some diversity in errors of coding, and to create different enough source programs to reduce the risk of common failures due to compiler errors or microprocessor bugs. However, the high-level DSDs may be more expensive and more difficult to apply correctly than the lower-level ones. There is an analogy here with the general issues in fault-tolerant design, of how to choose and combine between end-to-end fault tolerance (e.g. application-level checks and recovery actions) and low-level fault tolerance (e.g. hardware duplication or data redundancy), with their different coverages of different fault types and their different costs.

There is a general opinion that the really threatening problems in development arise from the high-level aspects of design: requirements specification, high-level specifications of algorithms; for threats in the lower level (conceptually later) phases of development we already have effective defences that should be applied anyway (irrespective of diversity) and are probably sufficient. On this basis, it would seem that the more useful DSDs are those applied earlier in the development process and higher in the design refinement hierarchy. E.g., functional diversity would be the most effective DSD; by contrast, a DSD like developing two different versions with C and assembly language would be much less valuable. We agree with the general thrust of the argument that the latter DSD should not be trusted much as a defence against high-level specification error, but we underscore reasons why DSDs in the later stages of development should not be discarded without proper analysis:

- system developers have minimal or no control over essential parts of the support platform, and high-level DSDs may be ineffective against their failures. E.g., in civil airliners avionics, microprocessor diversity is absolutely necessary as developers have to use off-the-shelf, complex, notoriously bug-ridden microprocessors;
- despite apparent high-level diversity, commonalities may re-emerge in the implementation of diverse channels (see examples in the discussion of functional diversity in the main text): one should explicitly check for such commonalities rather than trusting that they do not exist;
- methods for protection against errors in the lower levels of development, like the coding phase, do exist but are not necessarily applied, even in safety-critical developments. This defeats the premise that diversity is only needed against higher-level design mistakes. This is a special concern with COTS systems or systems "imported" from industry sectors with different certification and regulation cultures.

The process for choosing DSDs thus proceeds through these steps:

- analysis of threats (possible sources of human failures that will cause system failures);
- selection of DSDs by probable effectiveness on threats in the earlier stages of development;
- analysis of threats in later stages, and possible failure causes in the support platform, that may require further DSDs.

We also discuss how to deal with two further difficulties in decisions:

- one may need to trade diversity against reliability of the individual versions. Here, decisions are aided by analysing which weaknesses caused by DSDs can be obviated without reducing the positive effects of the DSD (section 2.7);
- forms of diversity are useful within the development of each version as well as between versions, and their combined efficacy will depend on how in detail they are used (section 2.8).

Discussion of individual diversity-seeking decisions (Section 3 and Appendix)

The preceding analysis is organised in a "top-down" fashion, starting from the general mechanisms of actions for all DSDs, and the general criteria for guiding their choice. Section 3 provides a "bottom-up" view, organised by categories of DSDs, and giving:

- a summary table of the mechanisms of actions, types of problems against which the DSD is a defence, and considerations about their cost and efficacy;
- an appendix with more detailed discussion and references to the parts of the "top-down" analysis (section 2) that support our conclusions.

Conclusions

Difficulties

A manager who wishes to employ diversity in a development project is confronted with a bewildering range of possible "diversity-seeking decisions" and their combinations. Despite the amount of industrial development effort and scholarly work spent to date on diversity, there is little guidance for these choices.

Massive obstacles stand in the way of better choices of methods for forcing diversity. Some of them will be reduced by additional research, especially combining existing understanding of human error with focused experiments, but we should not expect either simple cookbook recipes applicable to all situations, or simple algorithms that derive the ideal combination of DSDs from objectively measurable characteristics of a project. In this, the difficulties with diversity are not very different from those with any other choice of engineering methods. However, one can argue that diversity is especially difficult to comprehend intuitively, and so we will have special difficulties in applying our informal understanding to decisions about diversity. Indeed, many pages of this report are about ways in which intuitive understanding may be wrong.

Positive results

Despite all these difficulties, many coarse-grained "rules-of-thumb" about diversity can be stated, using general knowledge about software development as well as anecdotal evidence from applications of diversity. The role of modelling research in DISPO has been, for us, to clarify the circumstances under which the common-sense arguments that are common in the literature and in discussions among practitioners are applicable, which arguments are therefore inappropriate, and which evidence could support or definitely refute them. In a sense, this has created more questions than answers. This report offers to practitioners:

- a compilation of the "common-sense" considerations which drive decisions about the application of diversity, in the light of many years of research, including the insight derived from the DISPO project; and
- a more abstract analysis of the factors that determine the effectiveness of decisions in enhancing effective diversity and/or system reliability, to help readers in selecting among

decision rules proposed by others and developing their own decisions in their specific circumstances.

This report focuses on questions that have no purely mathematical answer and for which it is difficult to define clearly which answers are scientifically justified. So, this report, more than other DISPO deliverables, must be seen as representing the authors' informed opinions as experts in the reliability modelling of diversity. In addition to insight from mathematical modelling, we have tried to take account not only of the known empirical results, but also of the diverse expert opinions in the field. We have unavoidably described the wide uncertainty surrounding many decisions, but also outlined which decision criteria are based on more solid bases, and on which bases the more uncertain conclusions are based, so that readers can judge their applicability in their own situations.

1. Introduction

1.1 Scope of application of diversity and scope of this study

This study discusses measures for achieving useful diversity between redundant channels in a redundant system.

Diversity for achieving reliable systems is a very general approach with many possible variations. In particular, various architectures are possible for software-based systems that use diverse redundancy (see e.g. [Avizienis 1985, Strigini & Avizienis 1985, Strigini 1990, Lyu 1995]). The DISPO project has been mainly concerned with architectures that we may call "diverse-modular redundant" (Fig. 1.1), in which two or more diverse channels perform essentially the same function, seen at some level of abstraction. For instance, duplication with "conventional" "design diversity" implies two software programs that compute the same mathematical function (mapping from input values received to output values). Initially, the specified scope of the DISPO project was limited to simple design diversity in duplicated, diverse-modular redundant systems: two software versions perform the same plant protection function; the two versions are fed the same inputs and required to produce a command to shut down the plant under identical circumstances. "Parallel" actuation is assumed, guaranteeing that if either of the two versions correctly issues a shut-down order, the plant will be shut down. This is thus a 1-out-of-2 system, from the viewpoint of reliability of the shut-down function. The only aspect of interest was the reliability of the shut-down function, measured by the probability of failure per demand of the two-version system.

During the project, this scope was extended somewhat, to answer requests from the sponsors or to take advantage of possibilities opened by the research under way. So, some attention was also paid to how to extend the models for demand-oriented systems to continuous-control systems, and from "design diversity" to "functional diversity": two (hardware plus software) subsystems that perform the same system-level function, like sensing subsets of the state variables of a plant and ordering a safe shut-down when required, although using different subsets of state variables and different principles of operation. The limitation to "1-out-of-2" configurations was retained, given the central interest for the shut-down initiation function of protection systems.

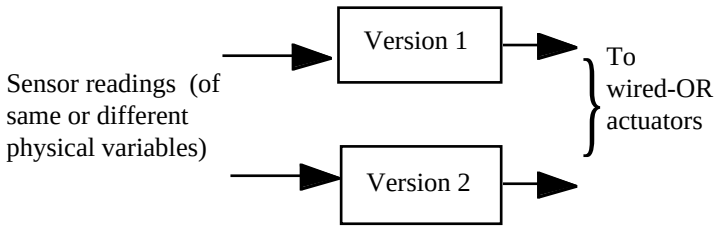


Fig. 1.1. Diverse-modular redundancy: example of 1-out-of-2 configuration (design diversity or functional diversity).

Diverse-modular redundant systems exist in many other forms apart from the 1-out-of-2, parallel actuation protection system. The main conceptual difference is in "adjudication", the process that determines the output of the diverse-redundant system on the basis of the outputs of all the versions. 1-out-of-N architectures are possible for protection systems, if either:

- the system has Boolean outputs and is built with wired-OR, parallel actuation, or
- each version has "fail-silent" properties: the version is designed to detect its own failures and then refrain from issuing active outputs towards the controlled system. The outputs are then processed via force-voting actuators or by picking one of the output values offered by the versions that did issue some output and can thus be assumed to be correct.

Apart from 1-out-of-N architectures, various forms of voting can be used; or the versions' outputs can be compared to detect version failures. For instance, in the Airbus flight control system, 2 versions form a self-checking pair, which can exclude itself in case of discrepancy, and two self-checking pairs form a fault-tolerant system.

For all these configurations, the goal in producing the versions must be a low probability of their failing together, in such a way that the adjudication function will choose an undesirable

output as the system's output. How to pursue this goal is the topic of this study. We do not study the adjudication function, which is also important in determining system reliability. Adjudication is reasonably well understood, and for applications like nuclear protection is very simple. For a given set of versions and a given chosen objective function (e.g., the probability of a correct system output), there will be a specific design of adjudication that (if implemented correctly) maximises that objective function (see [Di Giandomenico & Strigini 1990] for a survey of adjudication and a specification of optimal adjudication). Different forms of adjudication practically imply different redundant configurations. Most of the considerations that we make in this report apply to the whole spectrum of these configurations. [1]

There is another class of diverse configurations, "primary-checker" architectures (Fig. 1.2), in which the modules added for redundancy do not emulate the functions of the other modules, but have the purpose of detecting or containing their failures. These architectures include programs with run-time checks inserted in their code, combinations of modules which implement the intended function of the system with others that run reasonableness or safety checks on their outputs, auditing of a program's data structures (and possibly repair of damaged data structures using built-in redundancy), and so on.

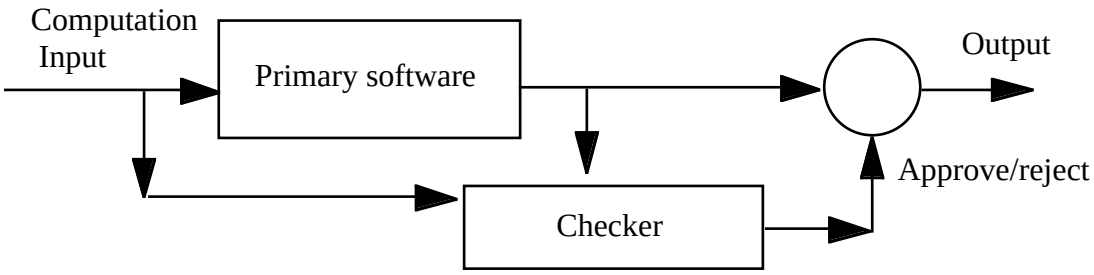


Fig. 1.2. Primary-checker architecture - basic concept

With these methods, it is still important to minimise the likelihood of common failure between the "checked" and the "checker" parts, but some of the considerations about achieving failure diversity that apply to diverse-modular redundant architectures may not apply to them. Diverse-modular redundant architectures are symmetrical, primary-checker architectures are not. One difference is that the "diversity" on which a designer depends is usually implicit in the application requirements; for instance, if the "checked" component computes a function that is easy to invert, a correctness check is readily available; if not, the designer may have to make do with less complete "reasonableness" checks. The checker and checked modules perform (often radically) different functions, which makes it appear easier to avoid mistakes leading to joint failures. Last, the designer may have more direct control on the failure probabilities of the checker: the probability of a checker failing to detect an error depends on its specification (how stringent a check is chosen), not only on the correctness of its implementation. For instance, in a primary-checker architecture the designer may choose to build an over-stringent checker, which reduces the risk of undetected failure at the cost of more spurious alarms, or an over-lenient one, with opposite results; he may trade off the effectiveness of a checker against its development or run-time cost. This degree of freedom about the stringency of checks for detecting errors in responding to demands is somewhat similar to that available, in designing channels of a protection systems, in choosing checks for detecting the demands themselves.

Many other architectural decisions differentiate architectures using diversity. Some of them may constrain the diversity achievable. For instance, a requirement for frequent cross-checking of intermediate results computed by two versions may impose a common structure to the algorithms they implement. However, these architectural issues and constraints do not affect the central questions about the efficacy of the various possible ways of pursuing diversity.

---

[1] The adjudication function itself has to be protected via redundancy and possibly diversity, of course. This creates further varieties of architectural solutions, but no special issue within the scope of this document.

1.2 Ways of pursuing diversity and how they relate to failure diversity

Software diversity, as initially proposed by Avizienis [Chen & Avizienis 1977], involves the development of functionally equivalent but independently developed versions of a software system. The term "independently developed" here meant that the versions are developed by individuals or teams that did not interact during the development process. It is hoped that, as a result, the versions (and in particular their faults) will be sufficiently different that, if they fail in operation, they will rarely do so simultaneously, and so an adjudicator, such as a voter, will be able to provide a correct output even in the presence of individual version failure(s).

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

The existing literature contain lists of such decisions that a design manager can apply to pursue diversity, which can be seen as "common-sense" advice (e.g., [Saglietti 1991, Lyu & He 1993] give developer-oriented advice; standards and guidelines, like [MoD 1996, MoD 1997] give customer-oriented requirements). For brevity, we shall call them "diversity-seeking decisions", or "DSDs". Quite often, in informal discussions of design diversity, the distinction is lost between "diversity-seeking decisions", i.e., what project managers can control, and the actual diversity achieved among the produced program versions and their failure behaviours, which we can measure *a posteriori*.

A complete list of plausible DSDs would span the whole development process, from team selection, to using different development environments, different tools and languages at every level of specification, design and coding, implementing each function with different algorithms, applying different V&V methods, etc. Some DSDs concern imposing differences between development processes, which can be specified regardless of the product to be developed. Others (like imposing the use of different algorithms) can only be specified with reference to an individual product.

However, the arguments in support of these various DSDs are mostly described at an intuitive level, and it is difficult both to trust them and to compare the relative merits of various DSDs. Most DSDs have a cost: duplication of activities, added co-ordination effort, need for staff with specific skills. The question becomes: how many DSDs are enough for us to achieve the desired level of assurance against design faults, or what is a cost-effective set of DSDs? There is currently no scientific answer to these questions, in the sense of either strong empirical evidence or a clear, predictive model of how DSDs produce their supposed benefits. A project manager or system-level designer will tend to rely on intuition, guided by personal experience. Experience is a poor guide for drawing general laws on how to avoid problems that are very rare in the first place. Intuition has been shown repeatedly to fail on these matters: the issues with diversity are subtle, and difficult even to define properly. For instance, some developers maintained that design-diverse channels would obviously fail independently, until this was proven wrong by theory and experiments alike. This report attempts to improve on unaided intuition by exploring in more detail the supposed mechanisms of action of DSDs in light of evidence from either general software practice or specific research on diversity.

Given our uncertainties about the relative merits of various DSDs, it would seem desirable to err by excess, by applying as many of them as we can afford. However, we cannot be sure that the advantages from various DSDs add up. We could think, for instance, that a DSD (say, specifying diverse algorithms for the various versions) produces benefits 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.

1.3 The various aspects of diversity and of "independence"

The only diversity that matters in practice is the *diversity between the failure behaviour* of the different versions. Informally, we want the failures (if any) of the different versions to occur in different circumstances.

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

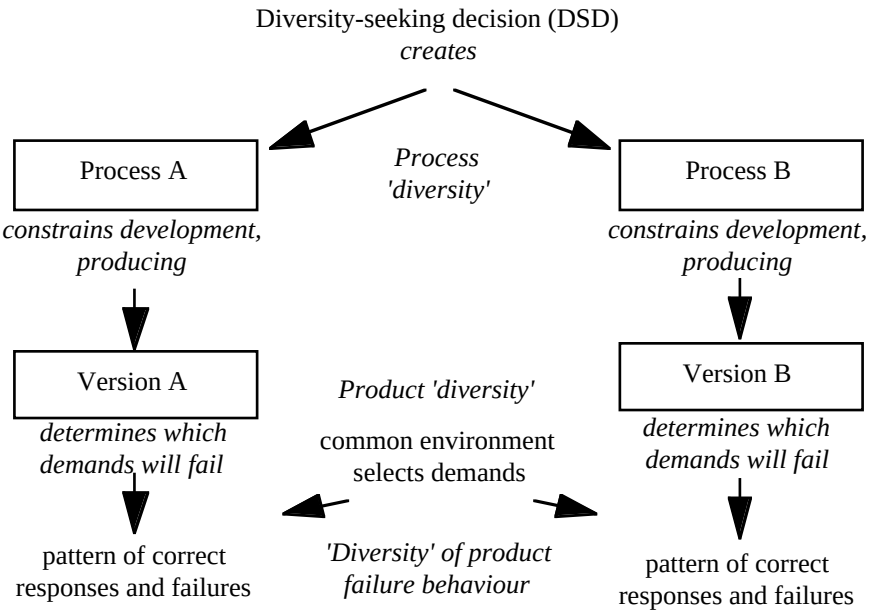


Fig. 1.3. The causal links from diversity-seeking decisions to failure diversity

The terminology used in describing such scenarios is affected by many ambiguities which cause confusion in analyses. A completely unambiguous terminology is unattainable and would be cumbersome to use, but it is important to point out the possible sources of confusion where they arise. Here, for instance:

- we say that DSDs cause "process" diversity. One may argue that many DSDs are really about *product* diversity, e.g. a DSD specifying two different algorithms for two versions, or mandating that two application versions shall run on different platforms. These are legitimate points; the important distinction to make is simply between diversity-related *decisions* (i.e., inputs to the development processes), which affect the way developers work (possibly by mandating aspects of the product they must develop) and the results of these decision in terms of *actual differences* between the delivered products or development, or between their failure behaviours;
- "independence" of failures between two versions means simply that the probability of the two failing together on a demand is the product of their individual probabilities. Other common uses of the word "independence" often cause confusion. Many practitioners and some standards (e.g., [MoD 1996, MoD 1997]) say that the concern in developing diverse

systems is to make sure that the versions are developed as "independently" as possible. This seems to call simply for the strictest separation between version developments, to protect them from any common influence (apart from the high-level requirements that they need to share). Actually, supporters of "independent" development of versions often support "forced diversity" - DSDs. Mathematical models [Littlewood & Miller 1989, Littlewood *et al.* 2001] confirm that "forced diversity" is better (in a specific, precise sense) than simple separation. Even with fully (statistically) independent developments, we should expect positive correlation between version failures. But "forced diversity" is, mathematically speaking, a way of introducing a certain form of negative correlation, i.e. of *avoiding* independence, between *development*, in order to *pursue* low correlation (independence or better - negative correlation) between *failures*. So, the phrase "independent development" often leads to confusion in these arguments and should be used with care.

1.4 Conceptual models of diversity

We briefly recall here the basic models that inform the discussion in this paper. An extensive explanation is in [Littlewood *et al.* 2001].

A common concept is that of possible *demands* forming a *demand space.* Each point (demand) in this many-dimensional space can be thought of as completely characterising a particular physical demand. For instance, for a reactor protection system a demand would be a vector of temperatures, pressures, flow rates, etc., sampled at regular intervals by sensor scans (the period of time required to define a 'demand' will influence the dimension of the vector). In this space, for a given program version containing a specific set of faults, some points are *failure points*: they are those on which the version will fail. We may group these points into *failure regions*, e.g. identifying a certain failure region as the set of all failure points that would be eliminated by fixing a specific defect in the code. The set of all the failure points (union of all failure regions) form the failure set of the given version. In a 1-out-of-2 system, the failure set of the system is the intersection of the failure sets of the two versions (Fig. 1.5).

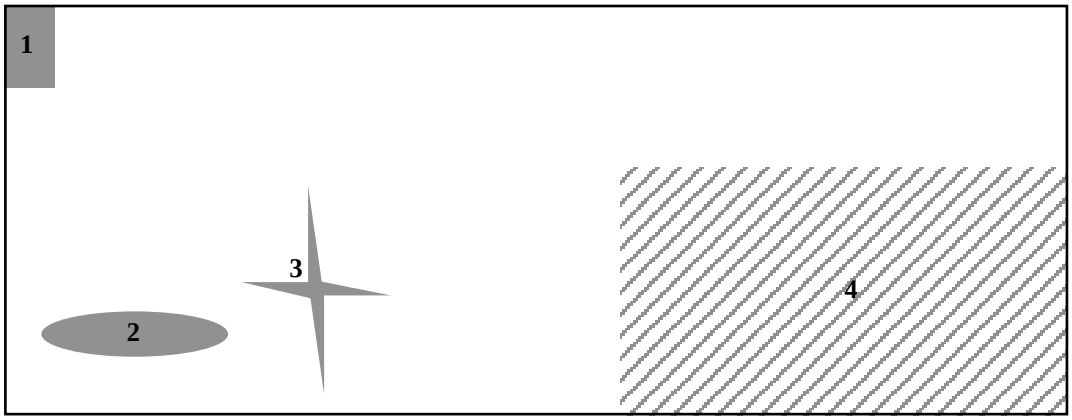


Fig. 1.4. Possible shapes of failure regions, represented here as grey areas in a two-dimensional section of a program's demand space. Analysis of actual defects in programs [Bishop *et al.* 1986, Ammann & Knight 1988, Bishop & Pullen 1988, Hatton & Roberts 1994] has shown that they may produce failure regions with simple shapes like 1 and 2 in this picture, but also complex "stars" or "snowflakes" or even non-connected regions like region 4 here.

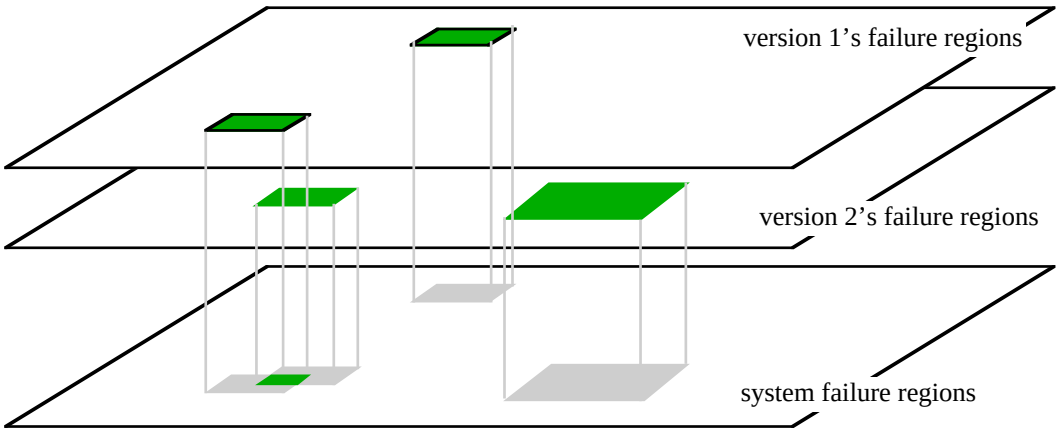


Fig. 1.5. In a 2-version, 1-out-of-2 system, the demand spaces for both versions and for the whole system are identical (and represented here by the three horizontal planes), but the failure set of the system is the intersection of those of the versions, as visualised here by projection: only the dark-grey intersections -on the bottom plane - are system failure regions.

The first conceptual model of software diversity was proposed by Eckhardt and Lee (EL) [Eckhardt & Lee 1985]. The key idea in the EL model is the notion of variation of "difficulty" over the space of possible demands. Consider, for simplicity, the case of a 1-out-of-2 system, involving two diverse versions. If an input is selected, and version A fails to execute the input correctly, what is the chance that version B fails, i.e. that the *system* will fail? The important point is that this *conditional* probability of B failing is then *greater* than the marginal probability of B failing, which would be the case under a naive assumption of failure independence. The reasoning is that the failure of A suggests that the input was probably a "difficult" one, and thus the chance of B failing is greater than it otherwise would be. The underlying informal idea is that designing software involves producing solutions to different problems, represented by different sets of inputs, and that some of the problems represent greater intellectual challenges (and thus scope for human error) than others. Consider, as an example, the case of a fly-by-wire aircraft: the inputs being received during a landing in severe wind-shear would be intrinsically harder to respond to correctly (i.e. would be intrinsically harder for designers of the various versions to program correct responses to) than those coming from straight and level flight in perfect conditions.

Littlewood and Miller (LM) [Littlewood & Miller 1989] generalised the EL model to represent the situation where diversity of design is *forced* upon the different versions, e.g. by stipulating that the different development teams use different algorithms, different programming languages, different testing methods, etc. - i.e., by applying DSDs. The useful effect of such forced diversity is that the "difficulty" of each demand will usually be different for the two developments, and vary differently among demands. If the variation is such that - informally speaking - the demands with higher "difficulty" values for one development tend to have lower values for the other, the expected reliability of a 1-out-of-2 system will be greater -everything else being equal- than if these variations are similar in the two developments. DSDs have the purpose of making this favourable situation more likely.

Clearly, some discretion needs to be exercised in practice over the extent to which diversity and thus particular design solutions are imposed on the developers. E.g., by mandating specific algorithms the specifications are enriched with more details than strictly necessary, thus extending the possibility of common influences of the (single) top-level specification activity over the development teams. These common influences may favour similar modes of thought which are prone to similar mistakes. Intuitively, the idea is to try to force sufficient "deliberate" diversity to decrease the tendency for coincident failures, without introducing novel opportunities for failure similarity via extra commonality in the high-level design.

Although DSDs can be forced upon different version developments to produce useful differences between versions, there are situations where similar differences between processes

occur *naturally*. Examples of such *natural diversity* include diversity in the programmers' domain knowledge, their fault finding strategies and other cognitively-related characteristics. These factors are often difficult to control, thereby making it difficult to use them as a basis to engage in a software diversity programme. The hope is that diversity in these factors should be significant enough to assist in reducing common mode faults. Further measures -DSDs - can be taken or practices can be instituted to enhance the degree of diversity between the versions. We next proceed to analysing them.

2. General analysis: how project decisions affect failure diversity

We begin by examining in general how diversity-seeking methods may actually achieve failure diversity. We will thus be able to discuss the individual DSDs in the list that will follow as special cases of general categories. This is important both to simplify the task of discussing them all, especially since the list is really open-ended, and to make it easier to *compare* the plausible effects of different DSDs, in the absence of empirical measurements.

Uses of diversity as protection against common mode faults in modular-redundant systems can be classified into three categories: data diversity, design diversity and functional diversity.

2.1 Data diversity and its mechanisms of action

Data diversity differs from the other two in that it does not require the channels of a redundant system to be themselves diverse. It is true that identical redundant channels would all fail when presented with an input sequence that caused one of them to fail. With data diversity, the designer attempts to present them with different sequences, with differences between them small enough not to prevent any of the channels from performing its intended function, nor their outputs from being in agreement (within some tolerance specified by the designer).

Data diversity may be achieved, e.g.:

- implicitly, via loosely coupled execution of the software copies and sampling of their sensor inputs (from different sensors or at slightly different times, relying on natural random differences between readings), or
- explicitly, by seeding small (possibly random) differences between the inputs to the components.

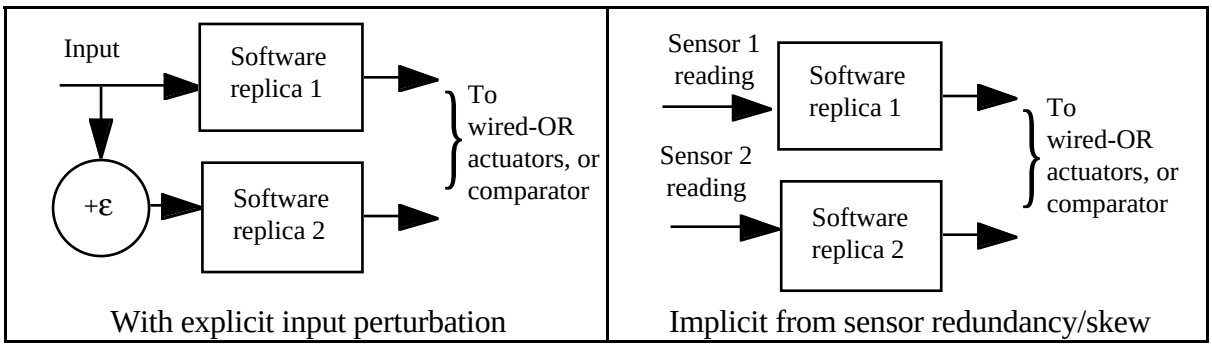


Fig. 2.1. Data diversity in duplex configurations

If the design faults are such that, of two similar input sequences, one will cause failure and the other will not, this can be exploited in fault-tolerant architectures:

- with two replicas of the software, comparison of their outputs allows detection of failures, and two such replicas constitute a self-checking pair
- with three or more replicas, it may even be possible to mask failures by voting, provided the failure regions are small compared to the distance between the diverse input sequences fed to the identical replicas of the software, because this would make it unlikely for more than one software copy to receive a demand that is in a failure region.

Data diversity as a project decision will be discussed further in Section A.1. We observe here that the feeding of slightly different inputs to two similar functions in two redundant channels (that is, "data diversity" as a run-time occurrence, whether intended by the designers or not) often occurs as a side effect of other design decisions. Examples are sensor redundancy, loose synchronisation between redundant identical channels (chosen perhaps to increase resilience to EMI-induced upsets), and other factors of complex behaviour in modern computer architectures, like interrupts, caching and dynamic instruction scheduling. These factors are often studied in the literature as a problem for the design of fault-tolerant systems: by disrupting "replica determinism" they may cause spurious disagreements between redundant copies of software, and complicate the design of state recovery mechanisms (cf [Poledna 1994]). A system design

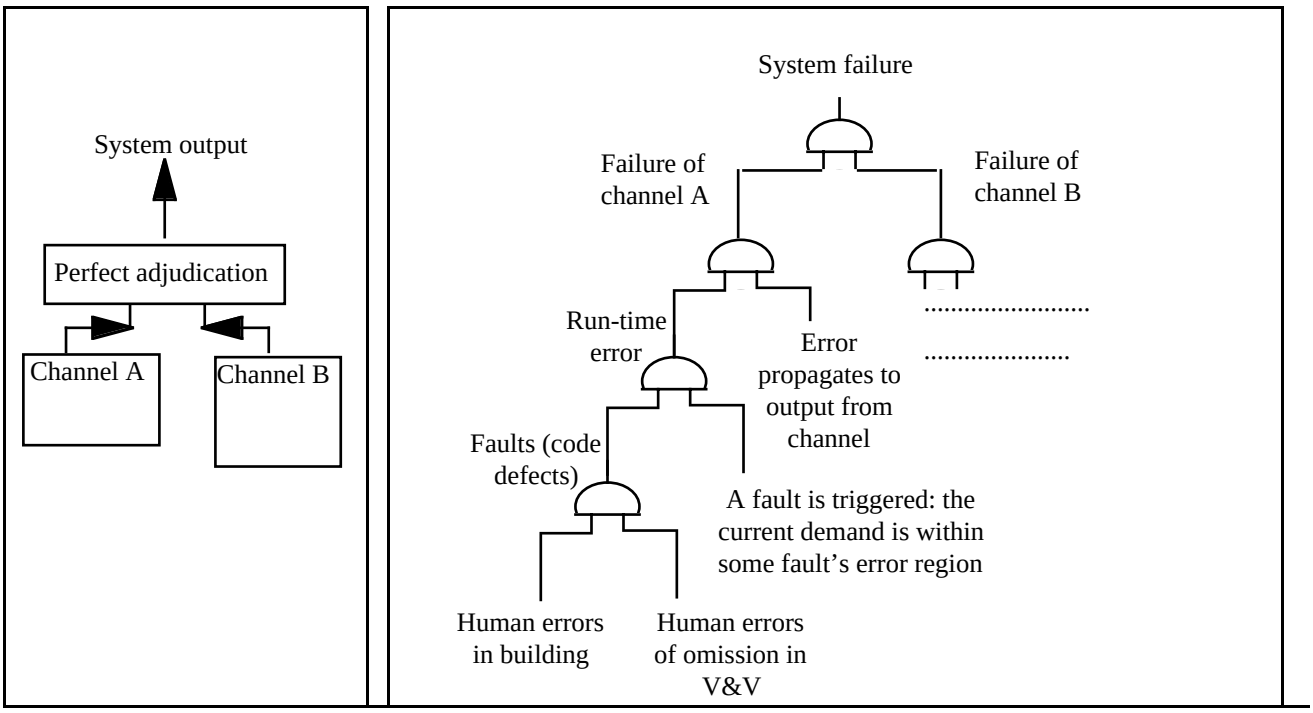
using design or functional diversity will also be affected by "unintentional" data diversity: for instance, if two diverse redundant channels produce an intermediate result through a numerical computation, even though code parts that nominally implement the same algorithm, they will often produce different results through the effects of numerical approximation. These differences will effectively appear as data diversity to the functions that consume these intermediate results: even if the implementations of these latter functions in the two channels suffered from the same conceptual mistakes, this "data diversity" would reduce the risk of common failure. The risk of spurious disagreements of course remains. This is a risk which the designers of diverse systems must consider in any case, though - it is useful to recall - it is less of a problem for a basic protection system (1-out-of-2 system with a single Boolean output per channel) than for continuous-control, majority-voted system.

2.2 Mechanisms of action of design or functional diversity

To examine how failure diversity may be achieved or enhanced, let us consider the steps from a development error to a common failure in a 2-version system in detail.

- developers (designers, inspectors, testers, ..) of a channel (version) in the diverse system make some mistake[s] so that a fault is left in the program. After the fact, we could describe the fault by asking the developers how the program created differs from the one that *should* have been created;
- on specific demands (the "error region" corresponding to that fault), the fault is triggered, i.e., the program behaves differently from what the developers would consider correct. The program reaches an *erroneous state*, in which some of its internal variables (including the program counter or other registers) has a different value from what was intended;
- either the error is masked (e.g., erroneous variables are overwritten before being used, the program terminates before they are used, they are used in an operation such that the erroneous value does not affect its result ....) or it *propagates* (i.e., causes further errors);
- if it propagates, it may eventually cause a *failure* of the channel, i.e., an erroneous output from the channel;
- if a similar chain of events in the second channel, possibly involving completely different mistakes, error regions and details of propagation, leads to the second channel failing concurrently with the first, the two-channel system may fail (or *will* fail, in the case of failure on demand in a protection system with pure wired-OR actuation).

This sequence is drawn in Fig. 2.2 in the form of a fault tree for a two-channel, 1-out-of-2 system.



System structure Fault tree for the event "common failure on a specific demand"

Fig. 2.2. Mechanisms of system failure in a 1-out-of-2, diverse system. (For systems where coincident failures of the two channels can be detected and tolerated, discussed in Section 2.3, a complete fault tree would be more complex)

If redundancy is applied by deploying identical copies of a program, all the steps in the list above will be identical between the two channels, except that data diversity might cause a difference starting from the fault triggering step. If instead we have process diversity, and thus some degree of product diversity between versions, differences may start at any stage of the causal chain:

1. mistakes may be different and cause code defects that are different; we could use the terms *human failure diversity*[2] and *fault diversity*; the two are distinct phenomena, and we would recognise them by subjective judgement, by examining the program versions and their development documentation;
2. we can hope that fault diversity will cause error regions in diverse versions that have limited or no overlap. This effect, however, may also happen without diversity of human failures or (probably) of faults either. For instance, two teams may both misinterpret a termination condition for a loop, but one of them may also use an extra condition that is sufficient to cause correct termination on most inputs. Conversely, even clearly diverse human failures may produce coincident error regions to exist for the diverse versions. Last, as discussed in the previous section, "internal data diversity" may occur: even if some parts of the code are so designed that they would err on identical data, the diversity between the parts that compute those data means that, for the *same* demand on the system, these code sections receive different data and the probability of their failing together is reduced;
3. error diversity causes *failure diversity*: two versions can fail together only if both have entered erroneous states. However, we may have failure diversity even if a certain demand triggers errors in two versions: these may propagate differently (including the possibility that one propagates while the other does not) because the two versions have different internal structures. Imagine, for instance, an overflow in writing to a data structure happens

[2] We are referring to human errors that are not corrected ("tolerated" by redundancy in the process), hence our use of the term "human *failure*" rather than "human error".

in two versions A and B, causing in both versions the overwriting of other variables. These variables may be ones that are immediately used in version A, but variables that will be overwritten before the next use in version B: when the overflow occurs, A will be likely to fail and B will not. Although the faults produce "error regions" that are the same subset of the demand space for both A and B, for A this is a failure region, for B it is not. It may also happen that both versions fail, but in different ways, e.g. with different results so that the failure can be detected.

These various forms of differences between events for two versions are summarised in the following figure. The arrows indicate causality. The open-tailed arrows indicate that "downstream" differences may occur even without the corresponding "upstream" difference.

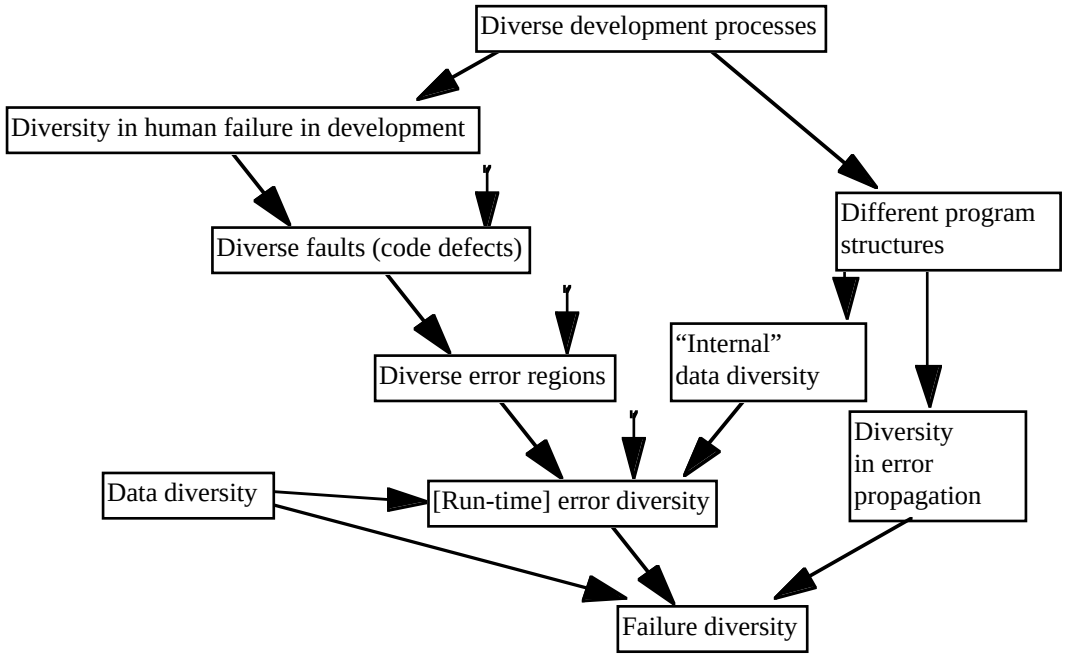
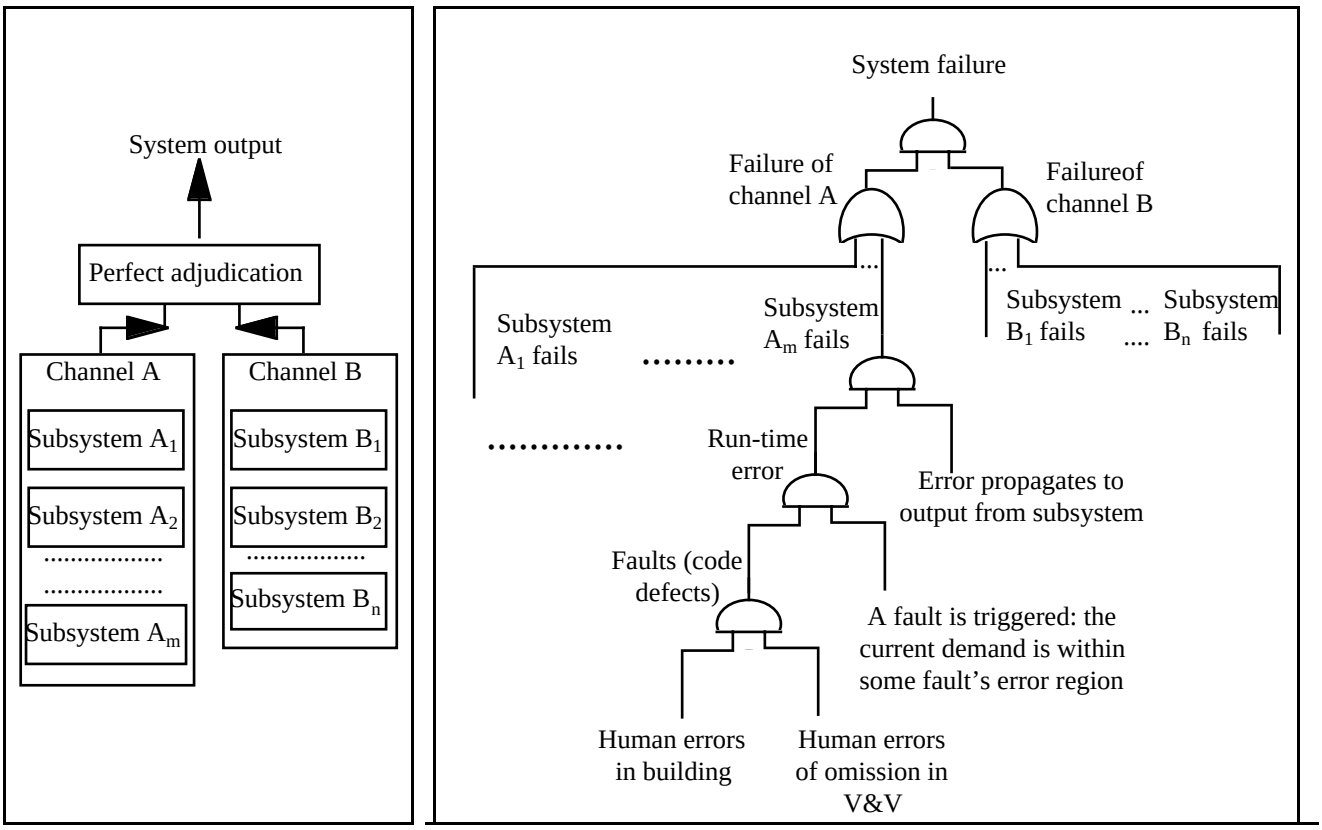


Fig. 2.3. A first sketch of the possible causes of failure diversity. Here (as elsewhere in this paper) "Development processes" includes both building and error-removal processes.

In practice, each channel may be made up of various subsystems, typically in "series" from the viewpoint of reliability. This is shown in the next figure. The channel failure sequence is only expanded for one subsystem in one channel. If the same sequence takes place in the other channel, system failure will ensue. Several complicating factors already appear in this scheme: for instance, the system may fail because of failures in the two channels that, although they happen on the same demand, affect different subsystems in the two channels (i.e., subsystems that perform different functions). And, of course, the two channels may also differ in the way they are subdivided into subsystems, so that there may be no obvious correspondence between subsystems in the two channels.



System structure Fault tree for the event "common failure on a specific demand"

Fig. 2.4. Mechanisms of system failure in a 1-out-of-2, diverse system: role of series subsystems in each channel

There is yet another way in which diversity may improve system reliability. A subsystem failure may be caused not only by design faults within it, but also by another subsystem failing. For instance, an application program may fail because the operating system on which it relies fails. Diversity between the implementations of a certain function (subsystem) in the two channels may be a protection against this kind of system (channel) failure, due to a fault *external* to the diversely-implemented subsystems.

For instance, a corrupted input value may cause failure of one version of a program, but not of another version which uses a different algorithm. An operating system (replicated identically in both channels) may erroneously shorten the time frame allotted to an application process for a certain operation, yet one version of the application may perform the operation within this reduced time frame while another does not. For want of a better name, we may call this *robustness diversity*. Robustness diversity differs from intentionally designed resilience, e.g., the ability of a channel (or a subsystem thereof) to detect pre-specified sets of illegal inputs, so that these may be better tolerated within each channel separately. With respect to intentionally designed resilience, we use diversity as a protection against mistakes in implementing it: so, if a channel fails to tolerate one of these illegal inputs or internal errors, the other channel will. By "robustness diversity" we mean, instead, unintentionally different reactions, in the two channels, to situations that are outside the design envelope (including both correct demands and those incorrect demands that the system is required to detect) for each channel or parts thereof.

With this addition, the whole set of cause-effect possibilities is as shown below:

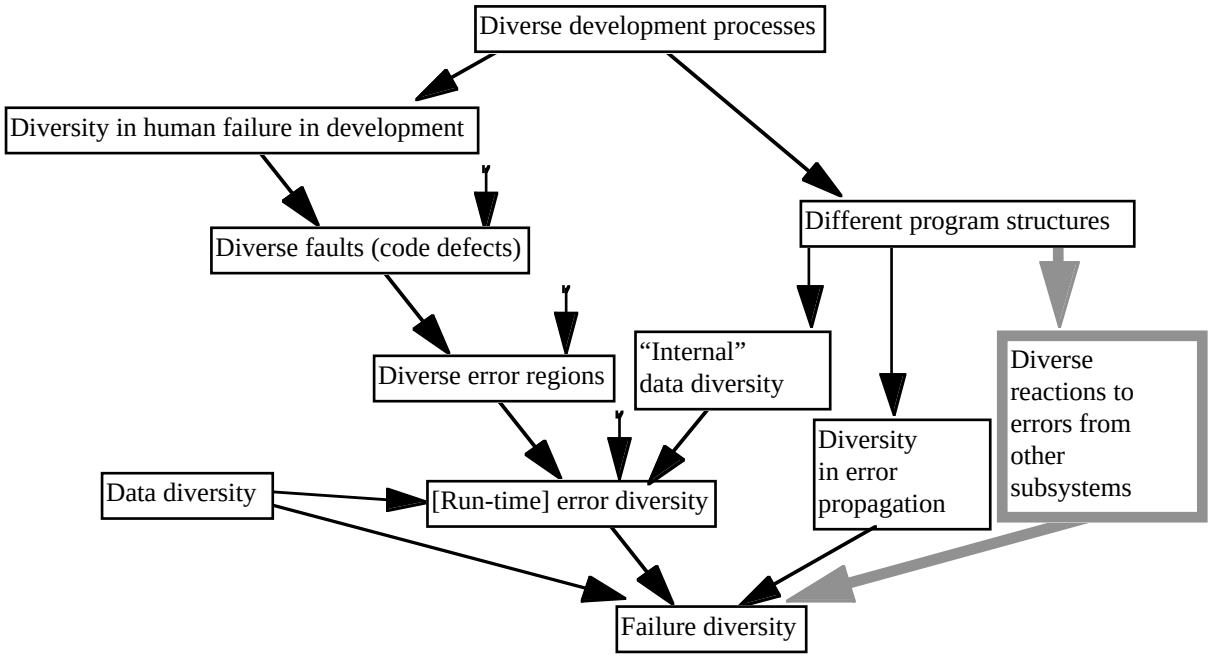


Fig. 2.5. Error propagation through diverse program structure as a cause of failure diversity.

Reasoning about diversity usually focuses on the main diagonal sequence shown in the figure: diverse developments cause different mistakes to be most likely for each different development team, and these different mistakes, if they are indeed made, are likely to create disjoint, or at least non-coincident, failure regions. This is the mechanism that is usually implicitly invoked in common-sense arguments in support of DSDs.

2.3 Failure diversity: separate failure points vs diverse failures on the same demand

There are two meanings of "failure diversity" or "diverse failure behaviour":

- *diversity in failure occurrence*: two versions tend not to fail on the same demands;
- *diverse behaviour during failures*: although two versions fail together on a certain demand, they do so with different (though both erroneous) behaviours, typically different output values.

The possibility of this second kind of failure diversity clearly adds to the benefit of diverse developments, although it would become irrelevant if the first kind were fully achieved, i.e. the versions had no common failure points. However, this knowledge has little practical use in design-diverse systems.

For a simple protection system having a single Boolean output ("trip" vs "do not trip"), diverse behaviour during failures is impossible and thus brings no advantage. However, even for systems with simple Boolean outputs it is often specified that the versions must produce certain common intermediate results, to be compared so as to improve failure-detection ability (at the possible cost of reduced potential for diversity in their implementations).

So, diverse behaviour during failures may offer an additional safety margin on top of that provided by difference between failure points, as it allows even common failures to be detected. When it occurs it allows, for instance:

- a 2- or 3-version system to be switched to a safe state (if such a transition is possible) by some simple comparator that is likely to work properly even in presence of common failures of the versions; or
- a 4-version system to tolerate 2 version failures; etc.

Predicting the extent of this aspect of failure diversity - the reliability gain it offers - is unfortunately even more difficult than predicting the first aspect: it seems to depend on details of the versions' implementation and their faults. Only a few aspects are clear:

- there is no possible gain if the comparison or voting takes as inputs a single Boolean variable from each version;
- the probability of a common failure being detected would seem to increase with increasing numbers of possible output values. In reality, what matters is the probability of erroneous outputs coinciding. In theory, given 10 possible output values, the probability distributions that are possible span the spectrum between these two extremes:
  - it may be that the typical faults cause both versions always to fail with the same erroneous value, so that there is no advantage at all over the case of Boolean outputs, or
  - perhaps typical faults cause errors such that the versions fail choosing each possible erroneous value with probability 1/9 and independently, so that the probability of undetectable common failure decreases by a factor of 9. This optimistic assumption is actually made at times, but is unjustified. In fact, if we do assume independence among the erroneous values (which seems optimistic), assuming them equiprobable is then again the most optimistic assumption possible.

  If we believe that common failures are mostly due to similar mistakes and similar faults, we should believe the first extreme scenario above to be the closer to probable situations: we cannot rely at all on the advantages given by the possibility of diverse behaviour during failures. Sometimes, the fact that the erroneous outputs may take many different values is used to justify a reduction by a certain agreed factor (e.g., 10 times) in the claimed probability of common failure due to physical faults, but we have not found any empirical or theoretical justification for such practices with regard to design faults.

Even when some DSD seems to improve the probability of diverse behaviour during failures, separately from that of diversity in failure occurrence, the former remains a more difficult issue and it seems unavoidable to take the pessimistic stance that one should not depend on this possibility in assessing the resulting reliability.

2.4 Human errors in development: random variation vs cognitive diversity

We have seen that failure diversity may depend heavily on developers of different versions making different mistakes. There are at least two mechanisms that may produce this "human failure diversity", due to the two characteristics of human errors:

- mistakes are a random disturbance in the development activity;
- yet, constraints on the activity affect the likelihoods of the various possible mistakes.

Because errors are unintended and unpredictable, repeating any human task in seemingly identical conditions will produce some differences in the errors committed. Hence the usefulness of "independent" (separate) development of versions. However, some errors are more likely than others: we may well observe that in performing a certain task there is a mistake that most people make. Against this possibility, simple "independence" does not help much: given a mistake (human failure in development) so common that the corresponding fault would affect, say, 90% of the versions developed, reducing this incidence to 81% via two-version programming does not seem very useful. No-one knows whether such high-probability faults characterise many projects. We do know that good projects tend to produce programs without frequent *failures*, i.e., that if there are such "90% faults" they have low probabilities of manifesting themselves as failures. But if we believed such faults to be common in high-quality developments, we should also expect limited advantages from "independent" developments: we should definitely try to "force" diversity.

Whatever the probabilities of each fault for a certain development process, another process may have different probabilities. Hence the attraction of using more intrusive diversity-seeking decisions than mere separation of developments. Their advantages are clear if we stick to the simplified view of the mechanism of action of diversity (the main diagonal in our figures): if we use two processes such that any mistakes that are likely with one are unlikely with the other one (human-error diversity), the probability of any one mistake affecting both developments will be low. If different mistakes cause disjoint failure regions, then this likely difference between the

mistakes will cause the two-version system to be much better than any one version and likely to have no failure point (with high probability).

Is it true that our diversity-seeking decisions can cause such sharp differences between the faults likely to be present of two processes applied to the *same* application problem? The answer is not clear. It is obvious that some development processes make some mistakes impossible. For instance, if one of two development teams is instructed never to use pointers, no pointer-related mistakes are possible. On the other hand, not using pointers means using a less powerful programming language, and will make some other aspects of software design more complex and thus error-prone. This looks like a useful form of diversity, but of course it will only be so in practice if the pointer errors likely for one team affect different demands from those affected by the errors caused in the other team by the difficulty of programming without pointers. Two extreme viewpoints are possible:

- with high-quality development processes, all important faults are due to basic difficulties in the application problem. Hence the dominance of requirement specification problems over coding problems, for instance. These difficulties cannot be changed by changes to the development process, which in practice affect the appearance of the problem, not its substance;
- the difficult parts of a complex development task are really the result of the combination of the problem to be solved with the strengths and weaknesses of the development process and team. Hence, changes to the development process, team and constraints really change the relative difficulties of parts of the task.

The truth, for any given project, must lie somewhere in between. The viewpoint that only high-level errors inherent in the application problem really matter is refuted by counter-examples from experience. E.g., the massive AT&T network outage in 1990 was triggered by "a `break` clause within an `if` clause nested within a `switch` clause" - a violation of a coding style standard. Its avalanche propagation mode was a system design fault, symptom of a lack of V&V procedures or run-time defences against it, but it would not have happened without the coding fault [Neumann 1995, p. 14]. Likewise, multiple telephone outages in 1991 were due to causes that included "a '6' instead of a 'D'" in the code, hardly a high-level conceptual error [Neumann 1995, p. 16]. Articles like [Lindner 1998] document that even simple procedures for code verification against such errors are not yet common practice. And the argument that high-quality processes eliminate those errors for which forcing diversity is possible is irrelevant for those projects for which such high-quality development processes are not practically available (due to the available staff qualifications, budgets, etc.), or for which "high quality" is not well understood.[3]

The viewpoint that the application difficulties can be changed greatly by altering the development process is supported by several pieces of evidence:

- psychological research has shown that even substantially identical tasks present greatly varying levels of difficulty, and hence different probabilities of errors, according to the way

[3] We should be aware that there are two meanings of "high-level" that are relevant here, and there may be confusion about which meaning we use. Psychologists classify the ways we perform mental tasks from a "lowest" level (by innate or learned reflex) to a "highest" (by actual conscious, structured thinking). Low-level errors ("slips"), like pressing the wrong key on a keyboard, are the most frequent and also those that we most commonly correct by ourselves [Reason 1990]. Organisations also have fairly common procedures for dealing with them. "Higher-level" errors usually result in decisions to do the wrong thing, while lower-level "slips" are failures to carry out a correct decision. In software development, on the other hand, the term "high-level" usually refers to activities or artefacts that deal with a less detailed view of a system and precede "lower-level" activities. So, "higher-level" software development activities like requirements specification can be affected by low-level errors (like typos), and "low-level" developments activities like coding can be affected by "high-level" errors - wrong decisions about solving coding problems. There is a similarity between the software engineering and the psychological classification of tasks, in that the "high-level" requirements specify an intention that the "low-level" coding must carry out; and for the coding phase we have more simple, automatable rules for checking for errors. But the similarities probably end here.

the tasks are described. Apart from laboratory examples[4], we are all familiar with the hopes raised by new programming languages that radically simplified development for some class of application - LISP, PROLOG, "4-th generation" languages, automatic control-oriented languages - by providing a way of programming that matched the way [some] developers instinctively "saw" the application problem. Notice that the simplifying effect of these languages is often culture-specific: the language may be natural for mathematicians, or for control engineers, but highly unnatural and error-prone for others. Likewise, a mathematically-oriented programmer may, for instance, feel at ease with both LISP or BASIC, and -all things considered - prefer LISP for a given task in view of its allowing a more concise, readable and verifiable representation; for another, LISP will still bring conciseness but at the cost of obscurity and thus be - on balance - more error-prone;

- at least for the more complex systems, only a minority of the developers actually face the application problem in its entirety. The task given to most developers is to create software modules according to some specification handed down by system-level designers, to test specific features, etc. For people developing utility procedures, for instance, the difficulty of their task is not related to which demands will cause the procedures to be invoked, but to how the various functions required in the system have been allocated to the procedures, affecting their individual complexity, numbers of arguments and so on.

If representation affects task difficulty, it will have the usual two effects: on the reliability of individual versions and on their diversity. If representation A makes a task less error-prone than representation B, why not choose A for developing all versions? Choosing A for all is indeed the best option if it makes *all* types of errors less likely; but software development is a complex activity, and a process characteristic that makes one task (or one aspect of task) easier may well make another one less easy.

When discussing the supposed advantages of diversity-seeking decisions we will thus often mention "cognitive diversity", meaning that a DSD may cause the tasks presented to the developers of the diverse versions to differ enough, either in their essence or in the way they are specified, to cause different error patterns.

There seem to be two important ways of creating "cognitive diversity": one applies within a single development task: e.g., adopting two different procedures for eliciting requirements, or programming a procedure with two vastly different languages, or starting from equivalent but very different representations. The other one is in the definition of the tasks: for instance, in subdividing a system into modules. The structure of a version of a software system generates a series of mappings from the demand space for the system to those of the version's subsystems. For the developers of each subsystem, the likelihoods of the various faults are determined by the specifications of the subsystem (and the tools and skills available). So, different subdivisions of the two versions into subsystems create the possibility of useful diversity in the mistakes made and the faults created. Likewise, the task of deriving the subsystems' specifications from the system specifications and the initial subdivision of the system into subsystems is a possible source of errors, and different structures for two versions give the possibility of useful diversity in these mistakes and the faults they could introduce.

This possible contribution to useful diversity is shown by the added arrow in the figure below.

[4] A famous laboratory example showed that playing the "tower of Hanoi" game became much easier if the standard different-diameter rings were substituted by different-diameter cups of liquid: observing the rule that one must not stack a smaller item on top of a larger one thus became obvious and no longer a demanding part on the subjects' mental workload. Another example is that playing "tic-tac-toe" is much easier than playing a mathematically isomorphic game in which people must select elements from a set of nine integer numbers, trying to produce a triple which adds up to 15 .

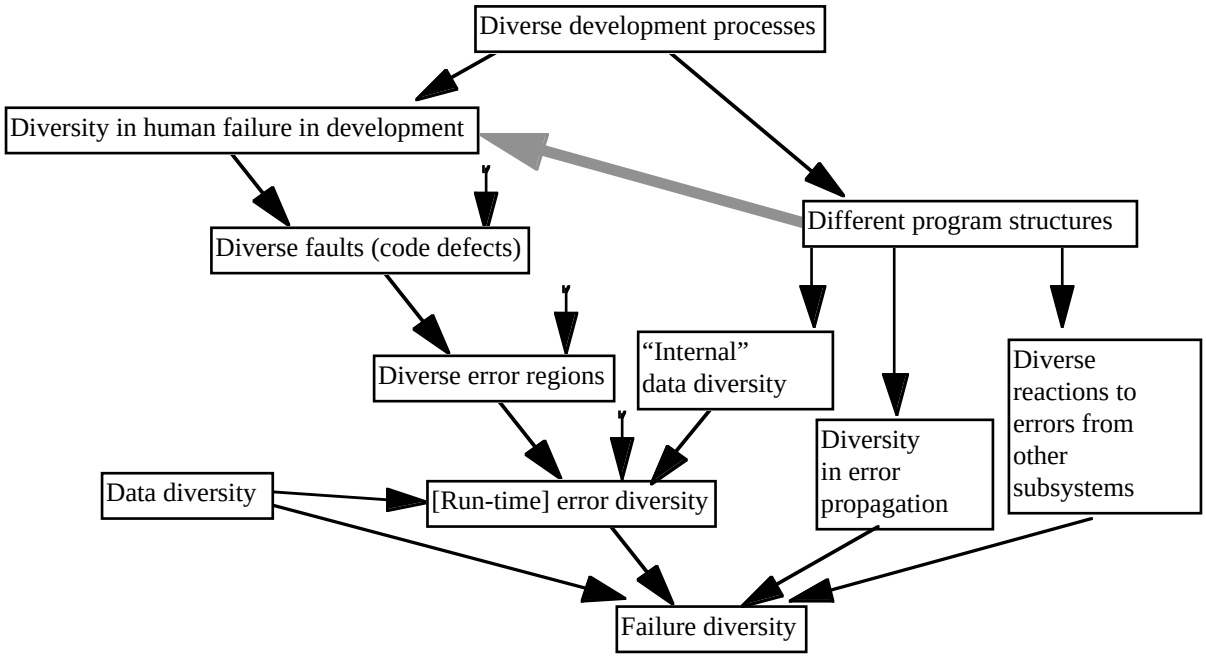


Fig. 2.6. Causes of failure diversity, including effect of program structure on human error

2.5 Difficulties in relating diversity-seeking decisions to failure diversity

This section delves into the causes of confusion that affect most discussions of forced diversity in the literature. We think this discussion useful as a defence against the common fallacies in technical discussions about diversity. In terms of practical advice for practitioners, this section can be summarised as saying that common-sense, intuitive judgement about forcing diversity should be taken with a large dose of scepticism: the suggested DSDs are usually likely to do no harm and possibly some good, but the arguments given about their utility for a specific project usually fail to prove that they *actually* do good. Readers who are not interested in the discussion of why this is so can safely skip the rest of this section. In the subsequent sections, we summarise useful criteria of judgement in the presence of this uncertainty and confusion.

2.5.1 The "mistake-fault-failure" cause-effect chain

The notion that diversity "propagates" down the fault-error-failure chain intuitively explains the advantages we hope to obtain from applying DSDs, but these cannot be taken for granted. The similarity of the terms "failure diversity" and "fault diversity" is misleading. Failure diversity refers to "a tendency not to fail on the same demands" (or "failing with different symptoms albeit on the same demand" - cf 2.3), and specifically high failure diversity means a low probability that both versions will fail equally on the same demand. Fault diversity is a subtler concept: it refers to a *tendency* of versions to exhibit qualitatively different faults. When we compare the faults found in two programs, we can decide whether we think they are different from some viewpoint, e.g. that they belong to different types, e.g., because they seem to be caused by qualitatively different mistakes (a programmer's typo vs misinterpreting the specifications) or they cause different failure behaviours (e.g., a memory violation trap vs taking a wrong branch in an if-then-else statement). We could even specify a measurement scale for the degree of diversity between two faults, or between the sets of all the faults in two programs. However, we are now interested in how a DSD affects the *potential for* diversity among the *unknown* faults that *may* remain in a pair of versions when deployed.

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

1. the difficulty of linking faults (defects in the code) to the specific demands on which they would cause failures;
2. the fact that the importance of a fault depends on the *probability* of those demands on which it causes failures. This is a real difficulty, but a comparably minor one. It is a real

difficulty, because even a DSD that were proven best at reducing common failures in a type of systems for a given regime of use might become useless if this regime changes. We don't know in practice the details of the demand profiles created by different system usage regimes, nor of the "difficulty functions" created by a DSD, so we could not anticipate which changes in usage would make a DSD useless. However, for highly reliable and simple systems, in practice an effective DSD will probably greatly increase the likelihood of two versions having no common failure point, i.e. zero failure rate irrespective of the use regime. So, if we could trust a DSD to reduce the probability of common failure points between versions, we would also generally trust it to reduce the probability of common failures. But this precondition is difficult to meet due to difficulty 1 above.

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

- as a code defect, defined by the difference between the faulty code and the code as it "should" be. This is the more natural way, but presents a difficulty. Two defects in diverse versions, which use different variable identifiers, and possibly different languages, will hardly ever look identical, even when they would cause common failures for similar reasons. So, versions may always have "different" faults, and useful metrics of "how different" they are become difficult to define;
- as the set of demands on which the defect causes the version to fail, i.e., points in the demand space that a given correction turns from "failure points" to "success points" (a "failure region" in the "demand space"). This is a less common view, but with it one can decide objectively whether two failure regions in two versions are disjoint, overlapping (and by how much) or identical, and from this information define measures of diversity.

Unfortunately, there is no general, intuitive law linking failure regions to the defects that create them. Defects that appear similar either in type or location may never cause failures on the same demands (they create non-overlapping failure regions), while defects that are different in appearance and caused by different mistakes may produce failures on the same demands (examples are e.g. in [Brilliant *et al.* 1990]). Moreover, software may have such a complex structure that even with a fairly precise understanding of a fault (as a defect in the code), it may be difficult to decide which demands will trigger the fault to cause a failure. An example is a fault in a deeply-nested procedure: we may be very clear about which arguments to the procedure will cause it to fail, yet unable to tell which demands on the system will cause the procedure to be called with those arguments, and even whether a failure of the procedure is guaranteed to cause version failure (cf e.g. [Bishop & Pullen 1989] for a discussion of "fault masking").

It was to solve all these difficulties that the EL and LM models introduced the radically new concept of a "difficulty function" defined in terms of probability that developers will cause failure, for each possible demand. This modelling stratagem acknowledges that all causes of difficulty in building programs combine together in complex, unknowable ways, but the end result can be described in terms of probability of failure per input point, and important insight can be gained from this measure alone. Unfortunately, this radical simplification makes the models unusable for our present purpose.

Since two versions may be such that recognising faults as identical between the two may be impossible, and yet a pair of faults (one per version) that affected overlapping sets of possible demands would of course cause system failures, we can only use experience of how DSDs affect "fault diversity" by referring to "types of faults", or "positions of faults" rather than to individual faults. We discuss briefly the meanings of these two concepts.

2.5.2 Fault "types" as a basis for reasoning about DSDs

Many schemes for fault classification are in use (e.g. the "orthogonal defect classification" (ODB) developed in IBM [Chillarege 1996]) in software engineering, so that statistics are available about the frequencies of the various types. Whether these data can be used to help in choosing DSDs is yet to be seen: we need to classify faults into such categories that on the one hand faults of different categories tend to cause failures less frequently than faults of the same

category, and on the other hand the DSDs of interest promote "fault diversity" in terms of these categories (i.e., the products of the two processes created by a DSD differ in the relative frequencies of the various categories of faults we define). A classification of faults in order to match them to the DSDs that may plausibly be effective against them is in [Saglietti 1991], to which we will return later. This paper has useful considerations about avoiding similar faults, but there is no evidence that this kind of fault dissimilarity increases failure diversity.

Any general link between frequencies of different fault types and probability of various failures is difficult to establish. This does not mean that it could not be found to apply *a posteriori*. Suppose I could develop many high-quality versions of a simple program, and measure them extensively, finding all faults and on which demands each fault causes failures. I might well find that of the few faults that are present, those of a certain type affect mostly a certain part of the code, which is only used by a certain subset of demands; and those of another type affect mostly another part, and cause failures on another set of demands. Still, one would need very strong evidence to believe that this linkage is a general characteristic of software development (at least for a certain type of software, e.g. "nuclear plant protection" or "spacecraft control") rather than an artefact of this particular experimental setting.

2.5.3 Fault location as a basis for reasoning about DSDs

Different considerations apply to diversity in the *location* of faults. We can give a precise meaning to this term if we can create a mapping, at some (possibly very coarse) level of detail, between the parts of code that perform similar functions in the two versions. We could then in principle measure the frequency with which a certain pair of processes (a DSD) creates defects in the same parts of the two versions. A lower frequency of this event would generally be considered an indication of a more effective DSD. Intuitively, the reason for considering this form of "fault diversity" a desirable feature is that defects that affect "corresponding" parts of the code in two versions are more likely to cause failures on the same demands than defects affecting "non-corresponding" parts. Again, this relationship cannot be taken for granted, but is plausible and it may be possible to establish it empirically.

2.5.4 Mapping of mistake/fault/error/failure types to DSDs

For the reasons discussed, it is common to choose DSDs on the basis of which kind of threat they appear useful against. One can use a classification like that given in [Saglietti 1991]. Faults are classified according to several criteria: the phase of the life cycle in which they originate; the form of human error from which they originate; aspects of their manifestation as failures; etc. Then, forms of diversity (the distinction between DSDs and actual differences in products is not emphasised as we do here) that should plausibly be effective against these fault categories are identified. For instance, if I am most worried about errors in translating the product specification into a high-level design, a recommended remedy is introducing diversity between the design representations used for the two versions, e.g. primarily "data-oriented" vs primarily "state-oriented". Prioritisation between alternative choices is not covered. A more elaborate manual based on these principles is available in German [Saglietti *et al.* 1992]. There are difficulties with such general rule-setting. Any classifications of faults will look arbitrary, incomplete or impractical (e.g. with too many overlaps between categories) to some readers. The mapping between fault types and DSDs is in some cases obvious but useless (e.g. there is no doubt that data diversity protects best against faults which produce "small enough" failure regions; but many designers have no idea of whether these are the dominant threat for them), in others useful but dubious (e.g., the insight that a DSD should be useful against a certain form of human error may be profound, but the case for its effectiveness may be based on intuition alone). Nonetheless, some degree of this approach is unavoidable: we will reason similarly in this document. Its problems should be lessened by using it as guidance in studying a specific project scenario, with its own problem complexity, dependability requirements and organisation constraints rather than trying to formulate general laws.

2.6 Practical criteria for matching DSDs to required fault diversity

The first criterion for choosing DSDs is usually how effectively they seem to reduce direct common influences on the development processes of the versions. Although no form of

diversity in development errors or in program faults that we could statistically demonstrate guarantees, by itself, improved failure diversity, it seems plausible that two versions affected by what is conceptually "the same" development mistakes are *too likely* to fail together. Any defence against this risk is welcome. This argument is strongest for the basic DSDs: the decision to produce diverse versions in the first place, and to separate their developments as far as possible so that mistakes cannot simply "propagate" from one development to another. Beyond this, any DSD looks desirable if it reduces the chance of such identical mistakes, because the possibility of it actually reducing failure diversity through another mechanism, albeit real, appears to depend on implausible coincidences; we have no special reason for expecting it to manifest itself in our specific circumstances. The adoption of DSDs, therefore, must be limited by cost, practical incompatibilities between them, and any demonstrated negative side-effects. The first two factors make it necessary to prioritise between DSDs that appear desirable.

1. a common approach is to prioritise by *threat*: analysing the (demonstrated or feared) weak areas of current development processes, for which remedies other than diversity appear insufficient; and selecting DSDs that seem most directly effective against these specific threats. For this approach, DSDs are matched to fault types, as in [Saglietti 1991], quoted above. This often means choosing a DSD for a specific *development stage*: we identify a stage of development as especially dangerous and "concentrate our diversity" on it, with the aim of having two processes which have "diverse weaknesses" in the same stage;
2. the above procedure assumes that a DSD affecting a certain activity is good because it *allows* for different distributions of difficulties between the two teams. It would seem better to choose a DSD of which we know that it *causes* such "different distributions". We may have such knowledge (though usually not as a certainty) in certain cases. However, before spending much on such a DSD we should check how plausible it is that the category of mistakes for which the DSDs create "human failure diversity" actually map into different failure classes;
3. there is sometimes a case for choosing a DSD which diversifies weaknesses *between development stages*. For instance, we may consider programming a mathematical function in a conventional language like C or as an "electronic spreadsheet". For certain algorithms, the spreadsheet format may greatly simplify the task of high-level design: complex control structures in the C program are substituted by data-dependency relations that are easy to visualise. However, the successive phase of static verification may be much easier in C: for instance, where a C program has a short loop construct for which we can easily prove that it satisfies certain properties at each iteration, the spreadsheet may have a huge array of similar cells, which require repetitive, error-prone inspection, one by one, to prove the very same property. How useful this form of diversification of weaknesses is (apart from any other advantages/disadvantages of this DSD) depends on whether mistakes in the two phases affected (high-level design and static verification) are similar or not in the failure points they would leave in the product: if the sets of likely failure points are similar, this DSD may just change the stage at which each process is most vulnerable, but not the degree to which diversity between them reduces common failures;
4. the problem remains (as said in 2, above), in adopting an expensive DSD, of deciding whether the evidence we have of its diversifying mistakes or faults actually supports confidence that it will diversify failures. This difficulty is serious, although it will vary between projects. The most difficult cases are internally complex systems, in which much of the code serves most demands and the correspondence between demands and specific difficulties they produce in any part of the design is irremediably obscure. At the other end of the spectrum, we may have situations in which different parts of the demand space obviously correspond to different parts of the implemented system, and these appear naturally to be most vulnerable to different specific kinds of human errors and faults. For instance, we may have demand types in which the main challenge is that of timing problems and overload and others in which the main challenge is that of complex numerical algorithms. Then, it will be most natural to seek DSDs that create diversity among these specific fault types. However, if a system allows us so clearly to identify the threats against different parts of the system, we will often have ways of mitigating them before we apply diversity, perhaps by applying different specific methods to the various subsystems (e.g.,

different programming languages, or verification methods). Applying these mitigation measures will presumably improve system reliability but also destroy any clear cues for matching DSDs to specific fault types;

5. cost considerations would suggest to restrict the duplication of activities required by diversity to aspects of development where it is really effective. This may take two forms:
   - *specific life cycle stages*: the later the developments of versions start to be diversified, the lesser the additional costs. Unfortunately, diversification from the earlier stages probably pays off far more in terms of reliability. Once it is decided that a certain stage of development will be replicated, additional or more effective DSDs applied to that stage need not increase costs greatly;
   - *specific subsystems:* those parts of a software system which have more critical roles (or, in some cases, parts that are though especially error-prone) are natural targets for efforts at enhancing reliability, including, of course, applying diversity. This may allow savings by focusing the use of (expensive) diversity measures on specific subsets of large, complex diverse systems, protecting critical phases of operation, or critical software components. This possibility is limited because many safety-critical computer systems do not have effective reciprocal protection among software components sharing a computer: all software sharing a computer with a safety-critical component becomes equally critical. Exception exist, like the platforms used for Integrated Modular Avionics in airliners, e.g. the Boeing 777.

We continue to discuss the choice of DSDs in terms of the important classes of threats and the effects of DSDs applied at a stage of development on subsequent stages. In sections 2.7 and 2.8, we will discuss two other decision issues: the possible conflict between increasing diversity and achieving version reliability, and the allocation of a form of process diversity within, as well as between, the development processes of diverse versions.

2.6.1 General classes of threats; threats from the support platform

Mistakes and faults can be classified by extremely fine-grained (and confusing) taxonomies, but two broad distinctions seem sufficient to clarify many issues in deciding about DSDs:

- design faults in the designed version, vs design faults in the "support platform" which is not under the control of the developers of the current system. The "support platform" includes:
  - compilers, linkers and other tools whose faults affect the production of the complete, executable diverse system;
  - hardware architectures and components, run-time executive or operating systems and other run-time utilities whose faults affect the execution of the versions in the diverse system (libraries used as black-box off-the-shelf items can be seen as belonging to either set of "support platform" components);
- among design faults in the versions being developed, faults originated by mistakes at a higher (earlier) level in the design refinement hierarchy vs those at a lower level. E.g., conceptual mistakes in specifying system structure or algorithms, vs mistakes in coding individual procedures specified by these higher-level decisions, and down to e.g. typographical errors in writing an otherwise correct procedure.

We believe that diversity at the platform level is a necessity (in general - calls for exceptions should be supported by very rigorous arguments) whenever there is concern about "support platform" bugs, as, e.g., with COTS microprocessors. It is appropriate to refute here a possible argument to the contrary, which runs as follows: if we adopt diversity at "higher levels", e.g. in algorithms or in programming languages, this diversity will guarantee that any specific demand made by the controlled system on the diverse versions will cause different demands to the support platforms, and the mapping between the two sets (demands on the two versions systems and demands on the platform of either version) will be so essentially "random" that even identical platforms will be unlikely to fail together. As can be seen, this is a "data diversity" argument: platforms with identical design faults receive different demands and thus are unlikely to fail together. To prove that it is invalid, simply consider a "difficult" circumstance for the designers of platforms to deal with. For instance, difficult situations arise when procedure-call

stacks get full due to too many nested calls. This circumstance is likely to arise when external events requiring action are unusually frequent (overload, especially in event-driven designs). So, we should expect higher probability of the non-diverse platform failing on those demands that exhibit higher rates of external events. The two versions are affected by similar "difficult points" in their demand space, which, as we know from models, imply positive correlation between their failures (here, between platform-caused failures in the two channels).

On the same basis, it becomes obvious that even diversity in support platforms should not be expected to guarantee independence between their failures, but only to reduce positive correlation between them: the support platforms implement similar functions and we should expect the difficulty of the demands made on them by the versions to be positively correlated unless we have strong arguments to the contrary.

2.6.2 Choosing between "high-level" and "low-level" DSDs; propagation of the benefits of DSDs through stages of development

Many experts think that DSDs applied to the earlier, "higher-level" stages of development (e.g., requirements specification rather than coding) are the most effective and most desirable ones (cf also 2.4), despite the higher costs incurred by replicating the development activities starting from an earlier stage. We see two sound arguments supporting this belief, but we think there are exceptions in their applicability:

1. the later stages in development pose fewer threats that cannot be mitigated by current techniques (cf 2.4). This is probably true, although, as we discussed earlier:
   - the degree to which those techniques are actually applied should be verified in each individual development;
   - the system developers are usually defence-less against design faults by support platform designers, e.g. in the compilers, hardware, operating systems. Specific DSDs may be required against these (cf 2.6.1);
2. The beneficial effects of DSDs propagate to some extent to successive development stages. For instance, having substantial differences between the specified high-level designs of two versions will cause differences between the coding tasks for the two versions as well. That is, they cause similar effects at the coding stage to the effects which DSDs directly applied at the coding stage (e.g., imposing different programming styles) would cause.
   This coding diversity is desirable to reduce the risk of both versions suffering from coding errors affecting exactly the same demands. It is plausible that the difference between high-level designs is as good a defence against these common failure points introduced during coding as one can hope for, and any additional DSD affecting the coding phase directly may bring no further advantage. On the other hand,
   - in some cases, DSDs applied at the higher-level produce essentially equivalent specifications of what the versions should do in detail for any given demand (these DSDs are still a defence against *common defects* in these specifications, but not against similarities in the possible errors in implementing them). Then, there is a risk that some parts of the implementation tasks are more error-prone than others, and further DSDs meant to avoid these difficult parts coinciding in the implementations of both versions are desirable.
   - again, faults in the support platform[s] may introduce common failure points.

A central issue in deciding DSDs is thus which sources of common failures exist at the "lower-level", later stages of development and in execution that the DSDs already applied at "higher-level", earlier stages do not prevent effectively enough. A useful way of looking at this is to look at errors in "lower-level", later stages from the view point of how much they are specific to certain demands. Given a certain set of DSDs at the earlier stages, and thus some kind of diversity "propagating" to the later stage we consider:

- at one end of the spectrum, some errors are presumably not linked to any special difficulty in "what the version has to do": e.g., typos or other slips. For common failures due to these errors, the "propagated" diversity", even if it only affects the appearance of the tasks at the current stage, may be enough to achieve as much protection as can reasonably be achieved.

The risk that human failure affect certain demands with higher probability than others seem to be avoided by the fact that slips probably happen with the same frequency irrespective of the demand they affect;

- at the other end, some errors would clearly arise with greater probability for some required behaviours of the version, and thus for some specific demands: e.g., some timing issues may be left for the low-level design to deal with, and be the same for both versions even given different higher-level designs.

There are many intermediate cases and problems of detail, of course. One must ensure the truth of the assumption that slips are not correlated to demands. For instance, it would become false if the parts of the design task that cover some demands are, for both versions, especially rich in memory-taxing detail, or if they are produced under more difficult works schedules. Then, one would want:

- to change the DSDs at the previous stages to reduce these sources of common difficulties between the versions;
- or to adopt DSDs at the current stage that reduce these common difficulties: e.g., different design methods, each better suited for some kinds of demands for the first problem, and different schedules of work for the two versions for the second problem;
- or to reinforce the common defences at the current stage (cf section 2.7), e.g. additional inspections against typos, and less strict deadlines, respectively.

2.7 Trade-offs between diversity and reliability of the individual versions

We need to consider that in pursuing diversity we may reduce the reliability of one or both of the versions. As a plausible example, imagine an application domain for which we have a standard software development process, and we have to develop a software product for which there is an obvious "right" way of designing it. In the effort to increase diversity between two versions of the product, we may decide that developers of version 1 will use the standard process and the "right" design, but developers of version 2 shall use a "very different" process and design, which we specify by mandating ways in which they must differ from those used for version 1. By creating these artificial constraints we create difficulties for the developers of version 2: version 2 will probably be less reliable than version 1. In so doing, are we improving or reducing the reliability of the 2-version *system?* Or, in other words, is "more diversity" always a good thing?

Cases in which forcing "more" diversity reduces system reliability because one or both individual version[s] become[s] too unreliable can easily be imagined; but so can opposite cases. In our example above, we have assumed that artificial constraints make the development more error-prone. So, version 2 will be likely to fail on more demands than if it were created with the standard process. If enough of these demands happen to coincide with those on which version 1 fails, the system will be less reliable than if it were developed without forced diversity. But if these demands tend to be very different from those on which versions created with the standard process tend to fail, then our form of forced diversity may tend to produce much more reliable 2-version systems than either single versions or 2-version systems produced by the standard process, although version 2 will as a rule be much less reliable than version 1.

What is clear from theory is that in some circumstances "more diversity" gives higher system reliability. A result of the previous (LM) modelling work (see [Littlewood & Miller 1989] for more details) is that forcing diversity - by adopting a DSD - is certainly beneficial (on average) if the DSD creates two process variants that offer the *same* guarantees of reliability - more precisely, such that we have no reason for preferring one process over the other. Consider for instance a project that can choose between two possible development processes, A and B, differing e.g. in the programming language used. Suppose that experience has shown that the choice of language does not seriously affect program reliability. Having to develop a two-version system, the right choice is then to develop one version with process A and the other with B. However, if experience in previous, comparable projects had shown A to produce generally

more reliable programs than B, we would need much additional information to decide whether the two-version system should be an AA or an AB system[5].

So, given two processes A and B and having to build a two-version system:

- if we have no reasons for preferring one over the other, we should build an AB system;
- if A is better than B, we have the dilemma that the best combination for producing a two-version system may be AA or it may be AB; or even BC, given a third process C which is also worse than A but is "very diverse" from B.

In practice, the latter situation may be less confusing than it seems. We seek diversity by trying to pair processes or methods that have complementary strengths and weaknesses: it seems important to discriminate between necessary and unnecessary weaknesses. The former are those that can be eliminated without "watering down" the process' strengths, and eliminating them is generally useful even though in a sense it "reduces the diversity" between versions. The following example illustrates this point.

The C language differs from Pascal in its lacking strict type checking (and in other characteristics which we will neglect for simplicity). Is this a form of diversity that we should seek? If we add to a C development environment additional checkers that enforce strong typing, we are clearly reducing diversity. We are also probably improving the average reliability of the C programs. Are we increasing system reliability too? Presumably so, if strong typing is generally beneficial (i.e., it makes mistakes, and thus software failures, less likely). Then, C's weak typing is an *unnecessary* weakness, and we ought to avoid it if the cost is reasonable (though we may not know how much reliability we gain). However, imagine (as a second scenario) that strong typing is still generally beneficial, but it greatly increases the difficulty of programming some special routines (e.g. processing complex data structures). These routines are used for a few demands, on which therefore the Pascal programmers are at a disadvantage. If this is so, "protecting" the C programmers (improving the C versions) by stronger type-checking may actually *reduce* system reliability, if it does not improve the *system* reliability for the majority of demands with which the Pascal programmers deal well, but it weakens the only strength that C contributes. In this second scenario, weak typing has become a *necessary* weakness if we wish to preserve the strength associated with it[6].

So,

- in a safety-critical project, and within certain cost constraints, it will be natural to avoid all "unnecessary" weaknesses. We will then often be in a situation of "indifference", in which we have no reason for expecting one process to deliver better reliability than the other, and therefore forcing diversity is always better than not forcing it;
- there are still weaknesses that are known to be "necessary", i.e. associated to desired strength and such as to increase diversity between processes. For these, we need to estimate to some extent the measure of these strengths and weaknesses: for certain combinations, even rough bounds would clearly indicate the appropriate decision;
- the most difficult case, which cannot be excluded, remains that of processes for which we know of a difference in achieved reliability, and know nothing about their strengths and weaknesses except that they may well be very diverse and thus worth "betting on".

---

[5] A similar rule applies to all 1-out-of-N systems with N>2. There is also a rigorous measure of diversity between processes, e.g. to decide whether method A is "more diverse" from method B than from another method C, but this measure of diversity cannot be estimated in practice.

[6] One may criticise details of this example and propose, e.g., that each routine be programmed in the language that is best suited for it, or type-checking disabled for certain routines, etc. This example remains valid nonetheless. What these arguments indicate is that there is a way of making C's weakness an "unnecessary" one again: we can eliminate it selectively and leave it in those places where it is actually a strength. But "necessary" weaknesses do exist because all these special arguments have merit in certain cases, but are impractical in others.

2.8 Use of diversity within the development of each version

We now turn to the importance of some forms of diversity to improve reliability of a *single* version. For instance, "diversity" between the weaknesses of the construction and of the verification phases pays out: a V&V method which is better at spotting those bugs that are most likely to be present (i.e., is strongest where the construction method is weakest) may be better than one that appears better if bug-detection efficacy is averaged over all bugs.

In DISPO, we have studied this issue with respect to V&V activities. Previous research (e.g. [Shimeall & Leveson 1991]) suggests that different methods have different strengths and weaknesses. Combining different methods is then sensible. We have shown in [Littlewood *et al.* 2000] the conditions under which applying two different methods with similar costs is better than applying the better method twice. These conditions include the case of strong diversity (in a mathematically precise meaning) between the two methods, and the case in which they have identical efficacy on average (the same principle of "indifference" we described when discussing the trade-offs between version reliability and diversity). Interestingly, estimating parameters so as to use these quantitative models directly in decision-making seems much more feasible for decisions about "testing diversity" in a version than about design diversity between versions.

Many (but not all) forms of diversity in fault generation can be pursued in developing a single version. For instance, when we give two development teams two specifications documents, S_A and S_B, which are written in different formalisms though equivalent, we hope that any obscurity they may contain will affect different areas of the specification. But then, could we not simply give both teams both documents, so that the team can use one document for clarifications whenever the other appears obscure? So, the faults caused by accidental obscurity may well be avoided in both the developments.

These considerations prompt two kinds of questions, which we discuss briefly.

2.8.1 Two diverse versions, or one version with all the benefits of diversity?

Extending the argument just developed, if giving both sets of specifications to the same team is effective in avoiding both sets of possible faults due to defects in the specifications, could we just have one team develop a single, good version? Rather than letting different faults enter the two versions, and then tolerating them during operation by combining the two versions, we could just develop a version without either set of faults! We will have the advantages of diversity without its complexity and run-time costs[7].

That is, one might argue (an argument that we state in order to refute its general applicability), in general, if we develop two versions so that we can apply to each step of their developments different methods, chosen for their diverse strengths and weaknesses, it may be better just to combine all strengths into one, single-version development.

[7] A special, border-line case of one-version process diversity is that of developers building two versions, but then only using both for extensive (back-to-back) testing of one of the two, which will become the operational version. Here, if both versions are built to be suitable for operation, the only advantage of running a single one is reduced complexity in the run-time environment; while an assured disadvantage is forgoing the protection of diversity for all those demands that were not applied during testing and may cause failure in one version only. However, the "test-only" version is often developed as a "rapid prototype" or "executable specification", which may not be suitable for operation (too slow, inappropriate for target platform) but allow its development to be high-quality, or low-cost, or very simple (presumably less error-prone) or just "very diverse" from that of the "operational" version.

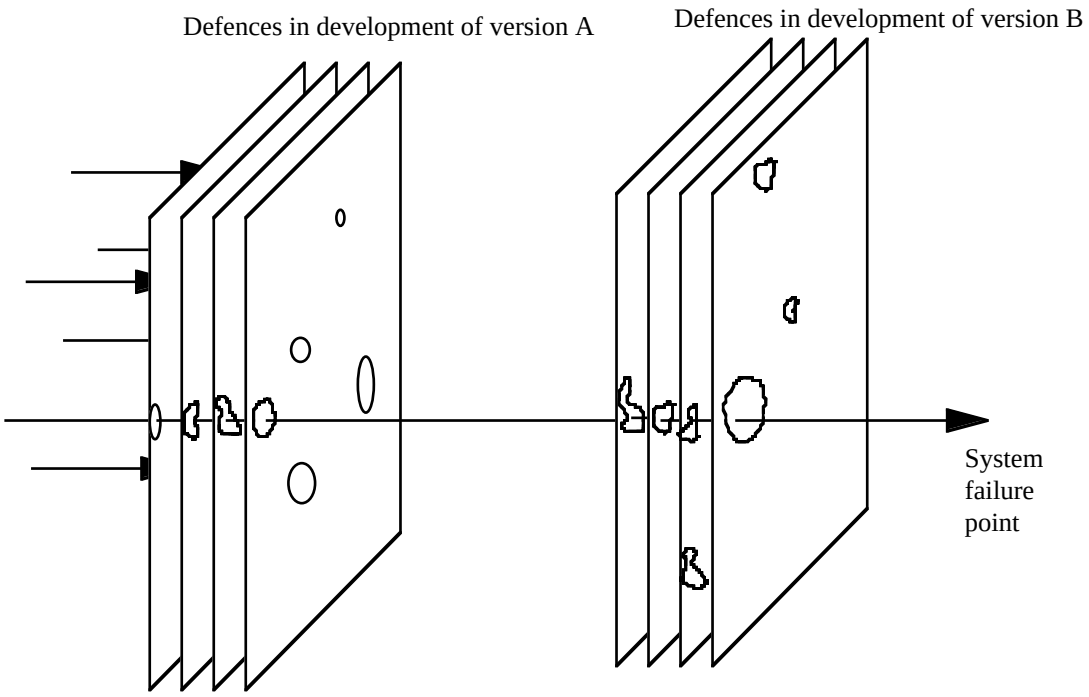


Fig. 2.8. Defences applied in development as a series of shields. The holes represent the weaknesses of the various methods. Only if there is a sequence of holes all aligned together, in the development of both versions, can a demand be a failure point.

This argument is best illustrated in terms the metaphor of development activities as defence shields with holes (or "Swiss cheese") in the figure above: if a supplementary specification document is an additional defence shield, it makes no difference whether this shield is in a position or another in the sequence of shields.

In terms of achieved system reliability, there are several objections to this argument:

- shields are not combinations of impenetrable parts and empty holes. Rather, their thickness varies from place to place, so that the likelihood of projectiles passing through takes many values between 0 and 1: whether a particular fault is inserted depends on a systematic component (the usefulness of the technique against that particular fault, represented by the thickness of the shield in that point) but also on a random component - whether a person will make that particular mistake at a specific moment. This is why even "non-forced" diversity is useful: even two identical shields offer more resistance to *all* projectiles than a single shield. So, using each specification document twice, if it is helpful at all in one version, is more helpful if done in two versions. However, it may be only marginally so, and then producing a single version with "specification diversity" would be practically as good as producing two (each using one of the two specification documents, or both using both), and cost less;
- more importantly in some cases, moving a shield, or adding others to it, may change its resistance. Here, having to depend on two specification documents may be for programmers a source of mental overload and added mistakes, i.e., make the single version worse rather than better. A decision must consider this possibility.

It is clear that the better decision depends on the details of the situation, and no general directive can be given. In some cases, experience and common sense will indicate an appropriate decision; in others, doubts will remain.

To illustrate the arguments that may come into play, we briefly discuss another example. When we order the two developers Andrew and Bernice to work on the two separate versions A and B, we hope that their different cultures and personalities make them prone to different types of errors. The faults left by Andrew in his version will then be different from those left by Bernice in hers. But then, cannot we simply employ both on the same team, producing just one, better

version by reviewing each other's work? The error points they eliminate will still differ, but they will be all eliminated in the same version: we will avoid faults "at the source" rather than just tolerating them in the assembled system.

In this case, the possible flaws in the argument are related to assuming that Bernice will be as effective as a reviewer of Andrew's work (or vice versa) as she is as an autonomous programmer. It is commonly accepted that additional inspections improve product quality, but that people are at times unable, when shown a ready-made solution to a problem, to detect some mistakes even though they would be unlikely to make those mistakes themselves in building a solution from scratch. Last, communication and management difficulties also matter: while turning two one-person teams into a two-person team may pose no problem, turning two 100-person teams into one 200-person team may be practically infeasible.

2.8.2 Best allocation of diversity across the development of a diverse system

The general decision issue is how best to allocate the "diversity factors" at our disposal in system development: e.g., people, forms of specifications, and so on. Assuming that we already decided to build two versions, we have still to decide whether, for instance, it is better to give both diverse specification documents to each development team, or one to each.

Again, there is no simple general rule, but we can specify the two aspects that must be considered:

- does this form of diversity, applied within a single development, increase version reliability? We may think, as conjectured in the previous section, that two specification documents will overload the programmers and reduce reliability. Then, we will obviously avoid giving them both documents. But perhaps we will think that if for a version we give specification document S_A to the programmers but S_B to the testers (making the testers "more diverse" from the programmers without overloading either) we will improve its reliability (compared to giving S_A to the testers too); we may then invert the roles of the two documents for the other version. Then, there is a case for this form of in-development diversity and the next question applies;
- when moving "diversity factors" around, could we be trading off version reliability against diversity and thus system reliability? Given for instance two verification methods, it may be still undesirable to apply in each version only one of the two: why settle for a sub-optimal process when we know a better one? We are back to the issue of reliability of versions vs diversity between versions, and to the argument for eliminating all unnecessary weaknesses, within cost constraints.

As a last example, the argument against "unnecessary weaknesses" implies that bringing individuals with diverse skills, experiences and viewpoints into the requirements, specification and validation teams is a good thing for both teams. Here too, however, there are practical limits, like the difficulty of co-ordinating many people or of maintaining morale and motivation as responsibilities are distributed and checks on people's actions multiply. Co-ordination difficulties may make in-process diversity completely impractical. for instance, it may be useful to procure two versions from two different companies, and we may be comforted by the knowledge that their personnel have largely complementary abilities. Mixing the same two sets of personnel into a one-version development process might appear to give just as good results in a mathematical model, but could be quite imprudent if the two companies had different internal cultures or different national languages.

3. Detailed discussion of diversity-seeking decisions (DSDs)

In Section 2, we have discussed the issues affecting the efficacy of DSDs and the factors to be considered in choosing them. This gives a framework for comparing the options available to a project, and a basis for a top-down decision process starting from general principles. In this section we give instead a "bottom-up" view, starting with individual DSDs. This is the way diversity is often discussed in the literature, and the way a project manager may have to organise decisions, starting from the set of DSDs that are realistically feasible for a specific project.

We start this section with a table which succinctly characterises and compares the many possible DSDs. In the Appendix we detail, for each entry in the table, the arguments, evidence and important exceptions concerning those DSDs, and the references to the supporting analysis in section 2. Each row in the table is labelled with the number of the corresponding section in the Appendix.

Preliminary notes:

- in the column "Probable mechanism of action, problems tolerated", it is implicit that the degree to which these types of faults or mistakes are tolerated is not usually known. This column is relevant for matching DSDs to perceived threats, and checking that all threats against which diversity is the preferred defence are actually "covered". Combining DSDs will generally "cover" the union of the sets of threats against which they are individually effective;
- in terms of efficacy against a specific threat, on the other hand, it is generally unclear how much combining two or more DSDs, that are believed to help against that threat, improves our defences in comparison to adopting just one of them. This improvement could be very limited;
- in the column "Considerations on cost, efficacy, practical experience", considerations of cost are limited to factors of additional cost caused by DSDs, omitting the obvious one that most DSDs require duplication (or n-uplication) of all stages of development subsequent from that to which the DSD is applied onward. In general, all DSDs carry a cost of replicated activities, an additional cost of co-ordinating the replicated activities, and savings in some activities. E.g., when testing multiple versions, it may be decided not to replicate the test data generation activity, executions must obviously be replicated, and the number of test runs needed to detect a fault may be reduced by detecting discrepancies among versions in back-to-back testing, even on test runs where failure of one version would not otherwise be detected. Having identified these cost factors, predictions of detailed costs should then reflect the cost structures of the specific organisations involved. Generic cost models have also been published to assist in these projection [Migneault 1982, Laprie *et al.* 1990, Voges 1994].

Table 1: Synopsis of diversity-seeking decisions

DIVERSITY-SEEKING DECISIONS	PROBABLE MECHANISM OF ACTION, PROBLEMS TOLERATED	CONSIDERATIONS ON COST, EFFICACY, PRACTICAL EXPERIENCE
A.1 Data diversity: - using random perturbations of inputs, or - using algorithm specific re-expression of inputs	Ensuring that if the input to one channel is within a failure region, the input to another identical channel may not be. It should be more effective with failure regions that are small or irregularly shaped	Generally cheap as no diverse versions required. Efficacy proven in experiments, very variable between faults. "Random" data diversity is often obtained *gratis* as a side effect of other decisions in fault-tolerant design. Special re-expression algorithm may imply additional costs
A.2 Design Diversity		
A.2.1 Separate ("independent") developments	Protection of developments against all *unnecessary* common influences that may lead to common failures	Most basic DSD, necessary pre-condition (and necessary cost) for applying most others. In some situations may be the most effective DSD.
A.2.2 Diverse development teams	"Forced" diversity via team selection is an appealing idea, but not proven in practice	Appears very desirable, if difficult to implement, *within* version developments; *between* version developments, serious issues of "diversity vs version reliability"
A.2.3 Diversity in description/programming languages and notations	Probable defence against some slips, and cognitive diversity against mistakes in higher-level problem-solving. Advantages affect both writers/verifiers of documents/programs and their users: implementors of next-stage, more detailed document. Diverse programming languages also usable to promote diversity in demands on platform	Efficacy must depend heavily on "how different" the languages are (e.g., functional vs imperative). With very diverse languages, issues of "diversity vs version reliability" With diverse specs, it may be possible to use appropriate language for each version and have very different languages
A.2.4 Diverse requirements or specifications	At all stages in development: cognitive diversity. Advantages affect both writers/verifiers of documents/programs and their users: implementors of next-stage, more detailed document.	Wide range of options, from purely aesthetic differences to specifying completely different behaviours, at system or subsystem level. The latter is presumably most effective, but increases system design effort
A.2.4.1 Different expressions of substantially identical requirements.	cf "Diversity in description/programming languages and notations"	Often cheap
A.2.4.2 Different required properties implying the same behaviour	Cognitive diversity benefiting both writers and verifiers of the specification and implementors of the specification Special cases: reduced-functionality secondary version, with scope for higher reliability checker-only channel, with greater scope for cognitive diversity	Applicability varies. Some problems may be easily specified via alternative, equivalent sets of required properties, some cannot. Issues of in-process vs between-processes diversity

Table 1: Synopsis of diversity-seeking decisions (continued)

DIVERSITY-SEEKING DECISIONS	PROBABLE MECHANISM OF ACTION, PROBLEMS TOLERATED	CONSIDERATIONS ON COST, EFFICACY, PRACTICAL EXPERIENCE
A.2.4.3 Requiring different behaviours from the diverse versions	Cognitive diversity benefiting both specifiers and implementors See also "functional diversity".	Applicability varies with availability of alternative algorithms for achieving same goal System design and version specification complications to ensure that diverse version exhibit consistent enough behaviour
A.2.5 Diverse development methods	Cognitive diversity benefiting those applying the methods and those applying its results	With complete, "packaged" methods (e.g. Booch vs Jackson) there is little chance of redesigning DSDs in detail: limitation but also probable savings. When designing differences of detail (e.g. applying different methods in requirement elicitation), issues of in-process vs between-processes diversity
A.2.6 Diverse verification, validation, testing	Both inherent differences in defects covered, and cognitive diversity	Issues of in-process vs between-processes diversity, "diversity vs version reliability"
A.2.7 Automatic code transformation	Producing different inputs to compilers to tolerate their faults	Cheap and obvious, but may be defeated by compiler optimisation
A.2.8 Diverse development platforms: diverse tools	Diversity in: limitations of tools in preventing mistakes; defects in tools that may cause software faults; presentation of problems to users (cognitive diversity)	A mixed bag of heterogeneous possibilities: wide range of costs, many practical constraints as tools are limited to applying specific methods and notations. In this sense, diverse tools may improve separation between developments
Diverse compilers (also applicable to replicas of a single version)	Diverse compiler bugs, so that any failure points they introduce are hoped to be different in different versions; diverse executable which may tolerate faults in run-time platforms	Usually cheap, popular due to compilers being in universal use, complex and known to have bugs
A.2.9 Diverse support platforms: run-time platform	Diverse platforms should exhibit different bugs possibly failing on different demands; diverse robustness w.r.t. application failures; possibly diverse requirements on application behaviour	Important as run-time platforms are known to have design faults and are often outside the control of application system designers. Also, platform faults may cause common failures on specific [classes of] demands irrespective of details of application
A.2.9.1 Separation and loose coupling. Diverse timing	Data diversity for both applications and platforms, in addition to better tolerance to upsets from EMI etc.	Usually decided on grounds of system design philosophy, hardware fault tolerance: advantages for software fault tolerance are *gratis*. With more complex adjudication than wired-OR, loose-coupled redundancy requires special care in design

Table 1: Synopsis of diversity-seeking decisions (continued)

DIVERSITY-SEEKING DECISIONS	PROBABLE MECHANISM OF ACTION, PROBLEMS TOLERATED	CONSIDERATIONS ON COST, EFFICACY, PRACTICAL EXPERIENCE
A.2.9.2 Diverse hardware	Different bugs, different compiler bugs	Comparatively inexpensive as mostly about buying diverse off-the-shelf components. Virtually mandatory as microprocessors (and probably complex ASICs) should be expected to have design faults. Cf practice in avionics for civil aircraft
A.2.9.3 Diverse operating systems or run-time executives	Different OS bugs; different requirements on applications, hence some cognitive diversity for application-level developers; different demands on hardware and thus some tolerance of hardware faults	There is evidence that even different COTS implementations of the 'same' operating system specifications (e.g. POSIX standard) exhibit some failure diversity. There is the attractive though unproven possibility of running application versions on radically different OSs, e.g. event-triggered vs time-triggered
A.2.9.4 "Partial" diversity, limited to subsystems	The subsystems that are diversified benefit from the effects of the DSDs applied to them; they may produce beneficial data diversity for the other (non-diverse) subsystems	May be a way of containing the cost of diversity, focusing resources on the more critical subsystems
A.3 Functional diversity	Cognitive diversity at all stages in development, and differences in limitations of sensors and physical models in regions of controlled system's state space May tolerate all kinds of errors, including specification errors and even gaps in understanding of controlled system's behaviour It should not be assumed to ensure failure independence between versions; common causes of mistakes causing common-mode failures may re-appear in later stages of development despite diversity at requirements level	Widely used throughout safety-critical applications to tolerate both physical and design faults. Intuitively appealing: maximum possible degree of cognitive diversity between developments, at the cost of developing separate specifications for the diverse channels; enforces separation of developments in later stages Increases system design effort to ensure consistency of top-level requirement on all versions (lesser problem for simple protection systems) See also "Requiring different behaviours from the diverse versions"

Appendix to section 3: detailed comments on DSDs

A.1 Data Diversity

Data diversity can be used as a stand-alone DSD, with identical copies of the software, or as an additional DSD in addition to design or functional diversity. In the latter case, it is usually the results of decisions primarily motivated by other reasons. We will mention these cases together with these other DSDs. The considerations about its probable advantages, however, are generally the same in all cases and we collect them here.

In trusting data diversity alone (input re-expression or differences between sensors or sensor readings) we trust that the differences we create between the inputs to two identical software replicas (similar arguments apply to configurations with more than two replicas) are large enough to give a low probability of both inputs lying in a same failure region, but small enough that both versions will usually produce correct and consistent results, to contain the number of spurious disagreements. The latter property is under the designers' control; the former is not, and we depend for it on assumptions on the sizes and shapes of failure regions.

Evidence about effectiveness of data diversity is somewhat sparse. Loosely-coupled replication is widely adopted, even without explicit consideration for design faults. In records of operation of Tandem fault-tolerant systems (with two copies of the software running on loosely-coupled computers), for instance, it tolerated [Lee & Iyer 1995] 82% of the software-caused failures: many software failures only happen in specific states of the operating system and application processes, which do not occur identically on the two machines. For intentionally-seeded discrepancies between inputs, Ammann and Knight [Ammann & Knight 1988] found that, on a failure-causing demand, retry with a slightly different demand would only cause a failure with a probability that varied, for different faults, from 0 to 99%. On the same principle, it has been shown that unreliable software may be made more reliable by frequent restarts ('software rejuvenation' [Huang *et al.* 1995]) which reset state variables of the system, purging them of erroneous values or other unusual, untested-for conditions they may have reached - this is also a form of fault tolerance. All these examples concur to indicate that data diversity would often be effective, but its effectiveness in a specific application is highly variable.

Further variations of data diversity require the "re-expression" of data to be constrained (though with possible elements of random choice) in relation to the algorithm implemented, to produce a special form of primary-checker architecture. E.g., if the software has to compute a function $f(x)$ of its input x, and it is known that $f(x)$ and $f(x+\varepsilon)$ satisfy a certain invariant $I(x,\varepsilon)$, we may feed to the second replica of the software a value $x+\varepsilon$ chosen with a large value of ε, to decrease the risk of x and $(x+\varepsilon)$ lying in the same failure region. The fault-tolerant architecture then takes the form sketched below. Ingenious examples can be found in [Blum & Kannan 1989, Blum *et al.* 1993, Blum & Wasserman 1994], although the probabilistic estimate proposed for the reliability increase they achieve may be optimistic, as based on implicit assumptions of independence between the presence of different failure points. Unfortunately, these methods can only be applied when we can first prove that useful invariants exist for the special algorithm considered. This is, in a sense, a boundary case between "diverse-modular redundant" and "primary-checker" architectures.

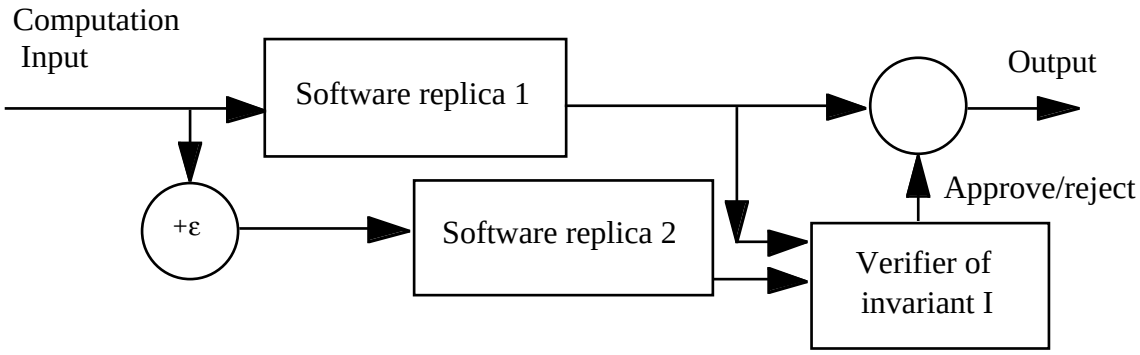


Fig. A.1 Self-checking software using input re-expression.

A.2 Design Diversity

Following common usage, we collect under the heading "design diversity" all those DSDs that can be applied to redundant channels which are specified to compute the same input-to-output function, although these DSDs can also be applied with "functional" diversity (which we discuss further on).

Terminological note; stages in development and development documents. All rigorous development processes used in safety-critical systems produce a hierarchy of artefacts, with some sort of requirements document stating "what the versions have to do" at the top end. Development entails transformations on the requirements to produce increasingly detailed design documents (variously designated as requirement documents, design specifications, design documents and so on) all the way to the source code, which in its turn can exist in different levels (e.g., application-specific language, C code, assembly code). In the discussion that follows, we somewhat arbitrarily divide these phases and the corresponding notations and tools into three groups: requirement specification, design and coding. The various sets of documents may be developed in a repetitive "spiral" fashion rather than in a monolithic "waterfall" process, but this definition of groups of documents remains valid.

A.2.1 Separate ("independent") developments

This is usually considered the basic requirement for diversity, and is pursued by forbidding direct communication between the development teams for different versions. Otherwise, it is felt, a misunderstanding within a team could be transmitted to the other one, and cause common development errors, faults and in the end failures. More subtly, a common perception could develop about what is the best solution for a certain design problem - e.g., allocation of required functions to program modules - which might well be indeed the best solution, but would reduce diversity among the developments in their subsequent phases, like the coding of the modules of the two versions.

A counter-argument here is often that more interaction would allow the natural diversity between human mental processes to improve both teams' understanding. However, this benefit of natural human diversity can to some extent be preserved despite strict separation between teams:

- each development team needs to have "enough human diversity", i.e., enough people and the standard procedures of inspection, testing etc. This will naturally happen on larger projects, but may be difficult to ensure in smaller ones;
- *some* (at least indirect) communication between the teams is unavoidable and necessary, but this can be mediated by project management to contain possible harmful effects on diversity. The necessity of communication arises if one team discovers an ambiguity or an error in the common specifications (which could cause common faults, or faults in one version that could easily have been avoided, or even spurious discrepancies between correct versions at run time). The management must then at times issue a correction to both teams. This problem was felt keenly during the "NASA-4 universities" experiment [Eckhardt *et al.* 1991]: the specifiers were application experts, while the programmers were not and required many clarifications. After each question, the experiment co-ordinators would broadcast the

question-answer pair to the programmers of all the versions, as an amendment to the specification documents. A long stream of amendments was thus generated. It was noted that this procedure risked propagating one programmer's perception of the problem to the programmers of the other versions: a better way would be for the project management, after answering each question, to decide which specification amendments should be broadcast to all developers, and distribute these as formal updates to the specs that do not reveal the reasoning of the specific programmer who asked the question.

It is interesting that Boeing [Yeh 1998] indicated these communication needs as the main reason for not using software diversity in the Boeing 777 (they did instead use hardware diversity among the redundant processors). This reference says that in the development of the 7J7 model the exchange of questions and clarifications between software developers and requirement owners was so intense that it "irreparably undermined the independence of the three teams". This is an example of the trade-offs between diversity and reliability of the individual versions: this intense traffic presumably improved the requirements and thus all the versions, and was more beneficial because multiple development teams were scrutinising the same requirements. It was reported in the PODS experiment [Bishop *et al.* 1986] that different interpretations of the same specs by two teams revealed specification ambiguities. By only considering "independence" (i.e., separation) among version developments as the desired goal of diversity, the Boeing managers were led to believe that they must discard diverse developments; had they considered that version reliability also contributed to the true goal of system reliability, they might have concluded that diverse developments were indeed the better option.

A function of separate developments is clearly to allow all the other kinds of DSDs, and their various effects of diverse processes as discussed in section 2. In theory, we could expect some diversity of effects even without separate developments. We could even re-use the same team twice to produce two versions; however, we know that members would be likely to remember how they saw the problem the first time around (or even how they solved it) and follow the same path, making it more likely that they repeat the same mistakes. There are cases in which diversity is entrusted to a single team: in defensive programming and primary-checker diverse architectures, it is often trusted that the diverse functions served by the primary software and the checking code are different enough to avoid common mistakes in their implementation phases, although we must observe to the contrary that the difference may not avoid common mistakes in the understanding of the models underlying both specifications. E.g., the specification of a file system manager and that of a file system integrity checker both rely on a common specification of the file system's redundant data structures.

The practice has also been recorded in which a team first develops by rapid prototyping an inefficient prototype of a system, and then the actual production version, and the prototype is used for back-to-back testing of the production version. This practice is usually a way of extracting additional benefits from the expense of prototyping, undertaken for a separate purpose like clarifying user requirements, not a way of pursuing diversity as a goal.

A.2.2 Diverse development teams

In practice, separate development teams always introduce some human diversity of cognitive and problem-solving style, backgrounds, skills. Forced, "designed" diversity could extend to the choice of staff, so that a certain role in one team is filled by a person whose main strength is in dealing with demands of class D_A, the same role in the other team is given to a person with more expertise on demands of class D_B. The practical difficulties in this approach are obvious, although it may be feasible in some projects. On the other hand, practices like having requirement reviews conducted by staff with diverse expertise can be justified on these grounds, and to an extent can be guided by considerations like these.

This is a case of possible trade-offs between version reliability and diversity, so the issue of unnecessary vs necessary weaknesses emerge. So long as we could afford it, we would tend, for instance, to provide multiple inspectors, with diverse skills, for design inspections on each version. Dilemmas may well arise here, for instance if we had only two inspectors that are expert (respectively) in two specific aspects of system operation: we could decide to dedicate one of them to each version, avoiding the risk of an inspector causing commonalities between the

modes of thought of the two development teams and thus increasing the risk of common failures, but giving up useful diversity within the development of each versions, and risking that some error be overlooked by one inspector that the other inspector would have picked up. If this latter risk were too much of a concern, we could even decide to share both inspectors between both versions, but we would try to adopt a communication protocol that minimises the risks of inspectors actually driving the developments of both versions. We would then be trading off diversity between the development of the two versions against that of diversity within each development (and thus higher version reliability).

A more extreme idea that could apply to diversity either between or within developments is that of choosing staff for their different personal characteristics - mental or generally personality traits - rather than just for different technical skills and backgrounds. Psychologists have identified, for instance, different problem-solving styles in people. One could try to recruit staff which exhibit desirable differences from these viewpoints. Experimental attempts [Westerman *et al.* 1995] have failed to show positive effects from this kind of diversity, while demonstrating positive effects of diversity in techniques used. This result does not discredit the idea completely, of course, but the lack of positive confirmation combines with the practical difficulties to make it less attractive. In addition, development processes for critical systems include many formalised procedures (e.g. structured checklists, HAZOP) which may either reduce the effects of such personal differences, or select for personnel with similar attitudes. Unless positive evidence emerges of its effectiveness, selection for personal and cognitive differences does not seem to be recommended.

A.2.3 Diversity in description/programming languages and notations

We discuss here the use of diverse notations for two versions, at any stage in development: specification languages, system design languages, programming languages. Many common considerations apply to all these cases. The main advantage to be expected is that of introducing "cognitive diversity" into the tasks performed using these languages: writing artefacts, inspecting and checking them, and understanding them to produce downstream artefacts, e.g. a program from a specification. As a minor advantage, notational differences may also help to prevent involuntary accidental communication between teams, as concepts may be difficult to translate between notations (on the other hand, some possibly useful notational differences are just restrictions on the use of a common notation, e.g. coding style standards or language subsets).

Important common considerations include:

- when the two languages are very similar (e.g., two forms of state transition diagrams for specification, or, among programming languages, C and Pascal) we would expect the only significant benefit to be protection against low-level mistakes: syntax errors that are common in one language but not particularly so in the other one (e.g., accidental use of assignment, "=", instead of equality "==" in C, much less common in Pascal due to different notations and because an assignment statement cannot be a term in an expression), and such. Whether this protection is significantly higher than allowed by simple separate developments (or possibly by more thorough V&V) is doubtful. On the other hand, if experience indicates that such an advantage exists, the cost of obtaining it is low;
- different advantages can be hoped from using together higher-level and lower-level languages, in the range (for programming, i.e., executable languages) from assembly-level to application-specific languages and executable specification languages: the advantage would come from shifting responsibility between application developers and platform/compiler developers. If a language provides built-in functions (or libraries) for doing something that would otherwise need to be programmed explicitly, the responsibility of avoiding mistakes in doing that is shifted from the program designers to the designers of the compiler and run-time support. We commonly recognise this as a potential benefit, in that compilers can be built to high quality standards as the cost is spread over many users (in some cases the opposite may be true, that compilers are less trustworthy than the present application's designers); but another aspect of this shift may be that different errors are common between compiler writers than are among application writers for a certain

application. Perhaps this difference would improve diversity in failures due to coding errors, but we do not see convincing arguments for supporting this claim. It may instead be the case that programmers working with languages of different levels may be in a better position to notice different kinds of deficiencies in the specifications from which they work: with a lower-level language, the programmer has to examine more implementation alternatives (e.g. in implementing data structures), which are instead pre-decided for the higher-level language programmers, with possibly important implementation details hidden and undocumented; the latter, on the other hand, may have more ease of examining the broad-brush architecture of the product;

- the goal of cognitive diversity between users of the language seems best accomplished by diversity between styles of language, e.g.: functional vs logic vs imperative programming languages, or data-flow vs control-flow based specifications. In specification language, useful diversity may be claimed between axiomatic specifications (e.g. algebraic), defining properties that must hold about the input-output behaviour of a program, and operational specifications, given in terms of operations to be executed.

  Such different styles of descriptions of problems or implementations may encourage different ways of seeing the problem, both at the level of small subsystems in a version and of a whole version. It should be noticed that seeking language diversity could impose a cost in reliability of some versions: some types of language will not be as suitable for the specific application as others. In an extreme case, it was noted in an experiment that programmers required to use PROLOG to program a control application ended up emulating an imperative language by PROLOG clauses: this was probably a pure loss in expected reliability for that version without any serious gain in diversity from the other versions. It is unclear whether a claim for effective cognitive diversity can be made for the difference between procedural and OO language. Perhaps the latter lead to a more data-centred view of design, the former to a more process-centred view; possibly, both views are possible with either style of language and their use for diversity should be encouraged with other means;
- for programming languages, another useful form of diversity is in promoting different lower-level implementations of the versions. For instance, block-structured languages use stacks more heavily than static-memory allocation languages; externally equivalent behaviours of two versions, implemented in an imperative language and in a logic language will probably be the results of very different sequences of machine instructions, memory access patterns etc., implying that similar demands on the versions are less likely to trigger similar design faults in the processor, compiler etc. (cf sections A.2.8-A.2.9). Different programming languages, of course, may also make it easier to use different compilers.

A.2.4 Diverse requirements or specifications

Development of diverse versions must of course start with a common system-level requirement stating "what the versions have to do". This applies even in the case of "functional" diversity, except that the described common behaviour concerns the effects of the versions' actions on the controlled system rather than the actions themselves (and the controlled system's states that trigger the actions rather than the way these states are detected). Development can start from a requirements document common to all versions, from which the recursive transformations into more detailed documents will proceed. At some stage in the development, diversity must be introduced, while preserving the initially specified degree of equivalence between the required behaviours of the versions. It seems desirable to introduce diversity as early as possible in this process (perhaps limited to subsets of the versions' subsystems or functions - e.g. protection functions but not control functions, if these co-exist in the same diverse system), as it is recognised that errors in the early stages are those against which even mature organisations have less effective defences. Faults due to omissions and ambiguities in the specification have been observed in several diversity experiments (e.g. the Project on Diverse Software (PODS) [Bishop 1988; Bishop *et al*. 1986]).

Two possible benefits can be identified from diversity in top-level specifications: diversity in errors in the definitions of the top-level specifications themselves; and introduction of cognitive diversity, from the very beginning, into the successive stages of version developments.

Several practical issues arise with diverse specifications:

- the different specifications must be equivalent, in a sense (specified by the higher-level system designers) which may vary from requiring bit-by-bit equality between their outputs, all the way to simple consistency with broadly-defined system goals. "Equivalence" here means compliance with both the requirements on the system that is to be implemented by diverse versions (e.g., implementing a protection function or control law), and requirements dictated by the need for the diverse versions to operate together. Equivalence is probably not a difficult problem for simple protection systems, but it may become an issue, for instance, with the introduction of cross-checks between the versions and the possibility that a looser form of equivalence causes more frequent spurious trips. These problems pose more serious constraints in control systems, where the diverse versions must: i) implement the same control laws within the approximation that would be required in a single-version system, but also ii) implement them in such ways that disagreements between versions are usually limited to cases of version failure (frequent disagreements among correct outputs would make adjudication and redundancy management difficult) and that the controlled system can tolerate the sudden change in the outputs of the control system that may occur when the adjudicator reacts to a version failure (the goal of smooth transitions at version failure can be pursued by appropriate design of the controlled system and the adjudicator, not only of the versions. The point we are making is just that the problem exists);
- so, the requirement specification for the multi-version system must include a specification of the desired form of "equivalence" between the versions' behaviours; and the validation of requirement specifications for the versions must include the demonstration of this restrictive form of equivalence (which may be difficult for some application problems). To verify equivalence, if two version specification documents are both derived from a higher-level formal requirements document, we can certainly check that both are faithful translations of it. But either the lack of this higher-level document, or the risk that the version specifications add details that would create unacceptable differences between the behaviours of the versions, may require some form of direct comparison of the diverse specification documents themselves;
- there is a trade-off between diversity and ease of error correction and detection. In a simple protection system, in which adjudication is logically a simple wired-OR connection of the versions' outputs, there is almost no constraint on how the diverse versions implement their common requirements. If, to improve error detection, it is desired to cross-check the versions' internal representations of the state of the controlled plant, it becomes necessary to specify relationships that these internal variables must satisfy at the moment of cross-checking. This constrains the freedom of the different developments to produce different designs, but perhaps not in a dangerous way, since maintaining these state variables may be a necessary part of any sensible implementation. Far greater constraints were imposed in some case studies implementing multiple-version, voted control systems [Avizienis *et al.* 1987, Eckhardt *et al.* 1991]: a common algorithm was specified in some detail for all versions. It has been claimed that in this last case no useful diversity can be obtained, and it is difficult to judge this claim: the diverse implementation did have different failure points, so that they were useful for fault tolerance, but it could be argued that those faults would have been eliminated by a higher-quality process, without a need for diverse coding;
- procedures for managing and controlling evolving specifications need to be established. Specifications tend to evolve, as errors and ambiguities are discovered and corrected during development. Keeping the specifications logically equivalent, particularly in a project with several teams, is generally perceived as substantially more difficult and expensive than for a single product. This additional difficulty may be even greater when products that are already in operation have to evolve, as there are additional design concerns that the specification changes must safeguard.

There are various ways of producing differences between specification documents, which we group into three categories: different expressions of substantially identical requirements; specifications of different required properties which still imply the same behaviours for all versions; and different required behaviours.

A.2.4.1 Different expressions of substantially identical requirements.

We can produce diverse, but equivalent, high-level specification documents by using different specification languages, e.g. ordinary English with various forms of mathematical and graphical notations, "formal specification languages" (e.g. Z). The general considerations of section A.2.3 apply here. We can decide to use notations which emphasise different viewpoints, e.g. data-centred vs state-transition centred. Alternatively, we could just rely on separate development of specification documents, possibly at the very top level from informal requirements. Whether this would allow greater or lesser "cognitive diversity" in the specification task depends on the particulars of the problem, people and methods concerned. In any case, it would complicate the task of checking equivalence of the two specifications.

Several experiments have used diverse specification languages. It appears that this, together with other DSDs employed - at least separate developments - tolerated some faults, but this is no clear basis for believing that diverse specification languages will produce substantially better reliability than could be achieved by simple diverse developments (see for example [Kelly & Avizienis 1983, Bishop *et al.* 1986, Bishop 1988]. Stronger, though anecdotal evidence that different specification languages can be quite useful in introducing cognitive diversity comes from an experiment on using diverse formal specification languages [McVittie *et al.* 1992]: the versions generated from one of the diverse (but equivalent) specifications tended to share some implementation details (not shared with the versions derived from other specification versions), although the investigators could not identify a direct cause of this similarity.

Diversity of specification languages can be applied in conjunction with either of the next two forms of specification diversity.

A.2.4.2 Different required properties implying the same behaviour

The two specifications may describe different required properties, which allow but do not prescribe identical internal behaviours. For instance, suppose that specification A describes a required condition R(i, o(i)) linking each input to the corresponding required output. Specification B might describe another such condition, R'(i, o(i)) which is equivalent to R. Or it might describe a series of operations on the inputs, which will in the end produce an output satisfying R. So, in general, the two specifications may be linked by the fact that satisfying specification B implies satisfying specification A (specification A contains fewer unnecessary constraints), or vice-versa, or both. We might even have that neither implication is known to hold: both specifications are believed to satisfy an "upstream" requirement (say: "the reactor shall trip when X happens", or "the credit application shall be classified as too risky if Y"), but their other implications are unknown (i.e., we do not know exactly in which range of circumstances spurious trips will happen according to either specification).

A.2.4.3 Requiring different behaviours from the diverse versions

We may require two diverse channels to exhibit different behaviours. In many cases this would produce what is usually called "functional diversity". We discuss functional diversity in another section, but the boundary between the two is not sharply defined, and both sections should be read in conjunction..

The specified differences in behaviour may affect internal behaviour only, e.g., the two versions have to run two different algorithms that produce equivalent results. This is a commonly adopted DSD, desirable whenever different algorithms exist for the function to be implemented.

An interesting class of specification diversity affecting the externally visible behaviours of the versions is as follows: one of the two versions ("primary") has a more sophisticated behaviour and the other ("secondary") a more rudimentary, but still acceptable one. For instance, the primary may offer better precision, better discrimination against false trips (if in a protection system), smoother control (in continuous control), etc. The "secondary" serves as a safety monitor for the primary, and/or a backup in case the primary fails. The failure diversity gains

that may be expected depend on the degree of "cognitive difference" between the two behavioural specification, but often the main goal is just higher reliability for the "secondary" channel, which is simpler to design, and has lower run-time resource requirements. In an extreme case in which the secondary version is very simple, so that there is a chance of believing it defect-free, through extensive verification or proofs, a sort of "argument diversity" may become feasible allowing a special form of independence claim between the probability of the primary *failing* and that of the secondary *containing any fault.* This subtle argument is developed in [Littlewood 2000].

A further development of this class of diversity leads to "primary-checker" architectures: the secondary version becomes a checker module, which no longer performs a function similar to that of the primary, but only acts as a watchdog on the primary, checking the correctness (when correctness tests are feasible for the function of the primary), reasonableness or safety of the primary's operation and outputs. These architectures are considered very attractive when the checker can be made much simpler (and thus probably more reliable) and/or cheaper to run than the primary. They should also be considered attractive for the degree of cognitive diversity they seem to allow between tasks in developing the primary and the checker: even if the checker were not expected to be more reliable than the primary, it would give better system reliability than a simple two-version system, as explained in section 2.7.

A.2.5 Diverse development methods

At all phases in the life cycle, from requirement specifications, steps of design refinement, coding, V&V, configuration control etc., the methods that are used can in principle be diversified. In practice, this is easier where possible alternative methods to use are clearly codified. E.g., the Jackson, Booch etc. methods are defined in reference books, as are more narrow techniques to be applied to specific phases of development, like "use cases" view or from a table-based notation.

Diversity in the development method may be essentially diversity of notations, discussed in section A.2.3, which may bring substantial advantages; but there may be more substantial differences in the activities performed, and these may be beneficial. Diversity in these methods is an intriguing possibility, especially in terms of producing "cognitive diversity", but little has been proven about it. On the down side, cost and availability of trained personnel are clearly serious issues. We give a couple of examples of how diversity of methods can be thought to help, with the warning that they are based on speculation, not on any empirical demonstration of the relative strengths and weaknesses of the methods:

- a "formal" development process - one which introduces "formal" specification notations early on - allows proofs of useful properties and forms of validation that are not possible in a development which does not use them. This is a strength of the method, but it may be accompanied by weaknesses: perhaps a risk that early proofs depend on assumptions that are inadvertently relaxed in the implementation, causing flaws the V&V process; or that the formal notations make it difficult for engineers to validate the specifications. Here we would have a case for imposing diversity between a process that relies heavily on formalism and proof and one that does not, in the hope that the weaknesses of the two tend to affect different classes of demands;
- "object-oriented" development presumably focuses attention on simplifying the inventory of code needed for a system, and avoiding integration problems through information hiding, while a procedurally-oriented development gives greater prominence to system behaviour. It is tempting to think that the two approaches will create useful diversity;
- the practice of specifying "use cases" as part of the requirement elicitation process may help completeness in defining the system behaviours required in some cases. It does not focus on completely covering the demand space. Thus, it is plausible that starting a requirement phase with use cases for one version, and with a more typically engineering-oriented method for another version, would give the two processes complementary strengths in analysing details of the required system behaviour in one case, of the demand space in the other. One may want to make sure that each development also applies the "other" method as well to validate the requirements, but even so, the different orders in which the methods are

applied may give the two processes complementary strengths and weaknesses in terms of the demands that can be affected by faults.

It can be seen from these examples that these forms of diversity are attractive, but:

- judging their effectiveness is difficult, because the strengths and weaknesses of the various methods are not well enough known (there is a case for more empirical study, which would benefit all software engineering, not just the engineering of diverse systems);
- therefore, decisions are affected by the problem of trading off diversity against reliability of individual versions (section 2.7). Since we cannot argue strongly that these DSDs will produce gains in system reliability through better diversity, we ought to ensure that the processes applied to the diverse versions are equivalent in the levels of reliability we can expect from them, i.e., we ought to eliminate any serious weakness that we perceive in a process compared to the others.

A.2.6 Diverse verification/validation/testing

Different V&V methods may offer two kinds of benefits:

- direct benefits, in that they differ in their inherent efficacy against different faults: e.g.,
  - "operational" testing tends to find faults with higher contribution to unreliability first, while "coverage" testing will have no such bias. So, for a given finite amount of testing effort, the sets of faults that (if present) are found by the two methods would be different. Formal proofs may find yet other kinds of faults
  - different formal methods are appropriate for describing different subsets of the properties of a program, and thus to detect different faults;
- indirect benefits, via cognitive diversity: different procedures may cause people to focus on different aspects of the problem, and of different strategies in the attempt to cover the problem exhaustively (cf the popularity of both FMEA and fault tree analysis, both of which, are just procedures for systematically covering the space of failures and their consequences, and thus may be considered, at a rather abstract level, as equivalent procedures).

Regarding testing in particular, there appears to be little experience of diverse testing to create diversity in the different software versions. One reason for this may be that it can be seen as a way of *deliberately withholding* one or more types of testing from a particular version (i.e. because they had been used on another version). Software developers might think this unduly constraining, and that it may result in individual version reliabilities that are lower than would otherwise be the case. It is likely that the efficacy of a testing method for detecting faults and improving reliability, for a single version diminishes with additional use. If this reduction in efficacy is rapid, it would be better (from the point of view of achieving version reliability most cost-effectively) to use a particular method for only a part of the allotted testing time, and then to switch to another method. On the other hand, using the same set of testing methods on each version in a set of diverse versions may not achieve the same diversity between versions as would using (say) a single (different) testing method on each.

The issue here is one of trading off increases in version reliability against increases in diversity between versions, and of how to best allocate diversity within and between version developments, which we have treated extensively in section 2.7. This issue requires further investigation (a work item planned for the DISPO2 project will extend the models of V&V diversity within one version - cf section 2.8 - to multiple versions)

A special case is that of "back-to-back" testing. This clearly offers a cheap test oracle and cheaper test generation than testing two versions separately. These savings may be important in some applications, so as to recommend back-to-back testing without question. However, if testing does lead to detecting and correcting faults, it will certainly reduce diversity between the versions. This will probably not matter as long as all the corrections are successful, but will tend to eliminate mostly bugs that do not produce common failures (or common identical failures). In detail: using comparison of the two versions as the only oracle means that we may mostly eliminate faults that did not threaten system reliability in any case (at least for 1-out-of-2

protection systems). Even with additional oracles, if we can afford few test cases and failure regions may be many and small, it may be more cost effective to provide different test cases for the two versions. We may eliminate more faults this way, and possibly more that have overlapping failure regions in the two versions.

Some of these considerations also apply to other components of verification and validation, such as walkthroughs and inspections, and formal proofs. However, the application of all these methods is both obviously constrained by available resources *and* dependent on other aspects of development. For instance, suites of development tools contain their own specific static analysis tools, which cannot be applied with other development methods. When such constraints exist, it is presumably desirable to select tools and methods in such a way as to enhance diversity. For instance, having to develop two versions, and to choose among three tool suites which appear equally suitable to support reliable development, but differ in the forms of V&V that they support, it would be reasonable to choose the two that appear to differ most in these V&V components.

A.2.7 Automatic code transformation

As a safeguard against compiler errors, hardware design faults and problems with arithmetic precision, there has been some limited use of automatic transformation of the source code of one version to obtain another, "diverse" version. For instance, arithmetic expressions can be automatically transformed by exploiting the commutative and distributive properties of arithmetic operations; Boolean expressions can be changed into equivalent expressions using the negations of the Boolean variables involved; conditional statements can be rearranged by re-expressing the condition ("IF A=0 THEN do x ELSE do y" becomes "IF A≠0 THEN do y ELSE do x"); statements can be reordered; one could (though this kind of transformation does not appear to have been used) convert arithmetic operations into equivalent ones by standard logarithmic or trigonometric transformations on expressions, and so on.

At least for simple transformations, these methods are very cheap and thus to be recommended. Their effectiveness is not fully proven with respect to real processor design faults, but they derive from time-honoured, empirically proven methods for detecting hardware faults by time-replication and comparison; so, we should believe that they are quite effective, at least if the processor design faults are similar to the artificial faults injected for evaluating these method against hardware faults.

A problem for these methods come from modern compilers' attempts to optimise code in such ways that all effects of these transformations may be cancelled by the optimisation steps. Then, this kind of diversity would only offer protection in those safety-critical development processes in which optimising compilers are still considered inappropriate, or only against errors in the front-end stages of compilation.

A.2.8 Diverse development platforms: diverse tools

We collect here some considerations that apply to all software engineering tools, although DSDs concerning specific tools are also discussed under other categories. Diversity is a concern with all software engineering tools because:

- they affect the way tasks are presented to people, and thus possible cognitive diversity between tasks affecting the relative likelihood of different human failures. Tool diversity is thus a way of achieving "notational diversity (cf section A.2.3) or method diversity (cf section A.2.5). Tools that support comprehension of designs and programs (e.g., behaviour and structure visualisation) also seem good candidates for diversity, either within or between development processes;
- they also have a direct role in preventing human errors or removing their effects on the developed program. If different tools cover different sets of probable errors, diversity seems useful;
- they are complex products, presumably never defect-free, and their defects can violate the intentions of their users and introduce faults in the developed versions. For many tools

(e.g., compilers, proof checkers), verifying their results directly is extremely difficult, so diversity appears to be a necessary precaution.

Diverse compilers

A category of tools that has received great attention in safety critical systems is compilers, as being, among automated tools, especially pervasive and complex. Compilers are known to introduce some faults, and diversity between compilers is likely to tolerate some of these. In addition, diverse compilations will often mean that the two compiled versions will not produce identical requests on the hardware, and will thus add a degree of protection against the ubiquitous design faults in complex modern hardware.

The use of diverse compilers is very cheap in terms of both money and time, and thus desirable although presumably ineffective against faults of the application versions proper. Diverse compilation is thus advisable even (or especially) in those modular-redundant systems where all channels run a single version of the software. Diverse compilations on *one* version can also be used, before deployment, to flush out via back-to-back testing or reverse compilation any bugs introduced by the compiler, and thus improve the individual versions (e.g. by avoiding the patterns of code that trigger the compiler's faults). Indeed, it would seem somewhat perverse to withhold from a version this kind of check.

A.2.9 Diverse support platforms: run-time platform

Diversity in the run-time platforms has received little attention in the literature on software diversity, but it is arguably the only form of diversity that is generally and absolutely necessary, as the system designers usually have no other effective defence against platform faults. Its effectiveness cannot be quantified yet: to our knowledge, there has been no empirical study of it; even studies of design faults in microprocessors [Avizienis & Yutao 1999] lack statistical data about design-caused failures.

We have argued before that diversity in the earlier, "higher-level" stage of development is not a sufficient protection against common failures due to hardware design faults, and hardware diversity should not be assumed to guarantee independence of these failures. Examples of stressful, "difficult" situations that are likely to be created for the support platform by the same demands on the system, even if applied to diverse versions of the application software, include demands with high rates of external events (discussed as an example in section 2.6.2), or other sources of overload (in operating systems, heavy requirements for communication buffers, processes, etc); floating-point problems, which have growing importance as it becomes more common to exploit microprocessor floating point arithmetics even in critical applications: failures (or exceptions, often treated poorly by compilers) are more likely to arise in some ranges of values of the controlled system's state variables.

A.2.9.1 Separation and loose coupling. Diverse timing

We recall here that increased separation between the diverse channels - separate sensors, loose synchronisation between sensor readings and between scheduling of internal actions - are ways of promoting data diversity between whole versions or subsystems thereof. This "data diversity" may, for instance, protect one channel from unusual timing patterns on which the other channel fails. It is often dictated by other design considerations than design faults, as it creates some degree of decoupling between the effects on the diverse channels of time-dependent disturbances: electro-magnetic interference, special situations in the computing platforms (e.g. transient overloads, buffer overflows, race conditions). Of course, it also introduces the potential for undesired discrepancies.

Redundancy without tight synchronisation requires special attention in designing communication and adjudication (e.g., insufficient care in implementing communication and voting among asynchronous redundant channels caused serious problems in the AFTI F16 experimental fly-by-wire programme), and some experts argue that the advantages are not worth the risk of design faults. But these objections do not seem to hold for simple protection systems. In any case the choice between synchronous and asynchronous operation is often made upstream of choices about software development.

Even if the diverse channels are part of a tightly synchronised architecture with a common time reference, timing diversity may be enforced by requiring that sensors be read at specified, different times.

A.2.9.2 Diverse hardware

The hardware of computer-based systems cannot be assumed to be free from design faults: it uses components (e.g., microprocessors, ASICs - application-specific integrated circuits) which are extremely complex, and many such components in the recent past have indeed been found to contain design faults. Especially with microprocessors, system developers may face the choice between using mass-produced, cheap, high-performance recent models and using much simpler chips (or even "formally proven" designs), with much lower performance and less of the assurance given by large-volume production.

It is thus clearly desirable to employ hardware design diversity, for the same reasons that call for using diversity in software. Diversifying the components that are considered more susceptible to design faults will to some extent create diversity in the rest of the design as well. It should be noticed that hardware design faults may be not just sensitive to input, but sensitive to history, time and environment (e.g., temperature or supply voltage) as well. It is thus possible for hardware to have many, low probability, design-induced failure modes, all of which can be tolerated to some extent by diversity.

Although the design faults in chips have received much publicity, one cannot exclude design faults in assemblies (boards, backplane-based assemblies, etc.), e.g. in the timing of bus interactions or the thermal design of boards. In hardware that is qualified for safety applications, these faults are likely to be rare and only triggered under special environmental conditions (ambient temperature, radiation levels), resulting in increased failure rates under special conditions: in other words, these faults, if present, will undoubtedly cause positively correlated failures in any non-diverse system. With respect to sensor design, this kind of considerations is already currently accepted as one of the motivations for adopting "functional" diversity.

A couple of less important, positive effects of hardware diversity are:

- hardware design diversity will also require diverse compilers, thus allowing some limited degree of protection against compiler errors. Indeed, modern microprocessors require increasingly complex compilers (to manage architectural features like pipelining, speculative execution and such; some newer designs depend on the compiler or assembler even to guarantee correct sequencing between operations belonging to separate instructions), increasing the risk of compiler errors and the desirability of protection against them;
- hardware diversity, even with software compiled from identical source code, is likely to create slight differences in execution timing between the redundant channels, and quite possibly in the results of computations (e.g., floating-point arithmetics). These effects may have the same desirable (and undesirable) results as explicitly-mandated data diversity.

If the executions in the multiple computation channels must be synchronised and/or voted, hardware diversity carries the cost of more complex hardware and/or design. Fortunately, this need not be the case in protection systems.

Hardware design diversity may be relatively cheap: it leads to an n-uplication of the project-specific hardware design costs; but, if the computers used are off-the-shelf units, these costs will be low compared to software design costs. For entirely bespoke hardware designs, of course, the cost of designing one hardware channel will be multiplied by the number of diverse channels required.

A.2.9.3 Diverse operating systems or run-time executives

It is common in simple, safety critical computing systems to have essentially no operating system. However, with increasing complexity of the application a layer of common support functions is desirable, which can be called an operating system or real-time executive: we will use the abbreviation "OS" to cover the whole category. Re-using pre-existing OSs is attractive in terms of costs and presumed reliability. Design faults in this layer can of course cause system failures. Hardware diversity will cause some differences in the behaviour of OSs, even if its is "the same" OS, ported to different hardware platforms, but these differences may be too

superficial to avoid common failures due to design faults. We see three directions for diversifying the OSs:

- OSs with equivalent functionalities but different origins. Here, the purpose is to avoid any common faults originated by common causes in OS developments. So, one would try to avoid, for instance, "ports" of the same OS to different machines, or OSs that share a common ancestor. The costs would be limited, and the interfaces presented to application versions would be rather similar. The more different the application interfaces, the more we could expect some additional advantage in reducing common failures due to errors in the designs of the application versions;
- OSs providing different levels of services. For instance, we may use for one version a basic executive that only provides time-slicing, and for another a more complete OS that provides inter-process synchronisation and message passing, scheduling and management of asynchronous peripherals. The effects would be similar to those described in A.2.3 for diversity between "higher-level" and "lower-level" languages;
- OSs implementing different architectural approaches: for instance, an "event-triggered" vs a "time-triggered" approach. Here, we could hope for greater advantages. Many design problems for the two OSs are likely to be different. The differences between OSs also impose serious differences in the way the application versions must interact with them, hence algorithm diversity, and probably in the whole way version developers have to visualise their designs. Moreover, it is a way of creating diversity in demands on the low-level platform (processor hardware) for similar demand on the versions, and thus improve tolerance of hardware design faults.

A.2.9.4 "Partial" diversity, or diverse execution environments for non-diverse subsystems

In a modular-redundant system, it is possible that, of the different subsystems comprising each channel, some are implemented diversely between channels, and some are not. Such a configuration does not pose any substantial new questions: the subsystems that are diverse can be considered as diverse systems in their own right, and create some form of data diversity for those subsystems that are identical (cf 2.6, item 5).

A.3 Functional Diversity

In functional diversity, a certain system-level requirement (e.g., emergency shut-down of a reactor under certain conditions) is satisfied by multiple redundant channels that implement different input-output functions: they typically use readings of different physical variables as inputs, and different algorithms, if possible based on different physical laws linking the state variables of the controlled system. Furthermore, the channels are often implemented in different technologies, e.g. software vs hard-wired electronics, although this possibility is being limited by the decreasing availability of non-computer based components.

Functional diversity has a long history in conventional reliability engineering as a protection against common mode failures. For instance, diverse back-ups to an electrical power supply might be both batteries and a diesel generator set.

Functional diversity is especially attractive as a protection against design faults because it may protect from errors in the functional specification of software, which could cause identical faults in diverse software versions. This is a difference of degree rather than one of kind, as we shall see, but it is still an advantage.

The possibility of applying functional diversity varies, of course, between different systems. Protection systems (where a pre-defined situation has to be detected), and other measurements systems where multiple sources of information exist (e.g. flight instruments) are typical applications.

High claims are commonly made for the effectiveness of functional diversity. In particular, it is claimed that it can be expected to achieve failure independence between the diverse channels. We have shown elsewhere [Littlewood *et al.* 1999] that this general claim is incorrect, for essentially the same reasons for which we cannot expect, in general, "conventional" design diversity to deliver failure independence: two functionally diverse channels operate on two

different transformations or projections of a common demand space (the physical state space of the controlled plant), characterised by its own varying "difficulty". This of course does not mean that functional diversity is a bad idea. It is plausible that it will often deliver better protection than pure design diversity, but we should examine under which circumstances the justifications for this plausible expectation hold. Since "functional diversity" is a general term for systems that have "more than just design diversity", we should analyse the various ways in which it is expected to be beneficial.

The use of diverse technologies in different channels is presumably a defence against common-mode *physical* failures (non-zero correlation between failures in hardware and software can be described by rather similar models [Hughes 1987, Littlewood 1996]. With respect to design faults proper, it seems to imply several DSDs that are considered beneficial with design diversity: different technologies often require different design notations and implementation techniques. Most importantly perhaps, there may be the advantage of cognitive diversity between the tasks of specifying the top-level requirements for the diverse channels: the mappings between the controlled system's state space and the demand space of each channel may be different enough to cause useful differences between these tasks.

Even for systems in which all channels are software-based, different algorithms are often specified for the diverse channels, either directly, or indirectly through basing their functional specifications on different physical laws. All the advantages listed remain, except the first. We have already discussed algorithmic diversity (cf A.2.4.3).

On the other hand, several factors may reduce the additional advantage that we expect functional diversity to produce compared to design diversity:

- the intellectual process of obtaining the diverse specifications from common system-level requirements is similar, with respect to human failures, to the process of writing programs from these specifications. We should examine how this task is made different for the different specifications of channels in the case of functional diversity, rather than assuming enough difference to produce substantial gains ([Burlando *et al.* 1992] attempts to model functional diversity in terms of differences between the "semantic domains" of the diverse versions);
- any uncertainties in predicting how the controlled system will behave in rare situations are likely to affect the specifications of both diverse channels - a concern that applies to most protection systems;
- for software-implemented channels, many needed algorithms (e.g., digital filtering) perform essentially the same function, irrespective of the input they process, and we should thus expect them to be subject to similar human mistakes in design, causing common-mode failures (e.g., both channels might behave incorrectly when their sensor readings vary along a certain pattern, and some reactor failure modes may produce this similarity in behaviour);
- last, there may be true commonalities in implementation: use of common software libraries, or libraries derived from common mathematical sources; similar processor hardware, sharing design faults that are likely to cause failures under similar conditions; etc. Some of these plausible sources of common-mode failures are not limited to software-implemented channels: e.g., filters with incorrect responses to specific input waves could well be a common-mode failure source in analog hardware implementations; application-specific integrated circuits are often designed by assembling pre-defined subcomponents from libraries and/or using CAD systems that may introduce common faults. What distinguishes software-based implementations is the difficulty of excluding such failure modes, due to the complexity of software and modern computer hardware.

In conclusion,

- it is likely that functional diversity will increase (possibly to a useful extent) the degree of diversity that could be obtained by "conventional" design diversity (applied to the same subsystems), although functional diversity should not be assumed to guarantee failure independence;

- this probable gain can be limited by common, system-level causes of common errors, e.g. difficulties in predicting stresses on the controlled system or its reaction to them, and by difficulties in applying truly different algorithms in the diverse channels; and may be limited or even nullified by commonalities introduced at lower levels, like similar implementation algorithms or similar platforms (hardware, operating systems, support software - *cf* 2.6.1, A.2.8, A.2.9). All these commonalities can be detected to some degree by analysing the specific system, and at least the latter can to some extent be mitigated by further DSDs.

References

[Ammann & Knight 1988] P. E. Ammann and J. C. Knight, "Data Diversity: An Approach to Software Fault Tolerance", *IEEE Transactions on Computers*, C-37 (4), pp. 418-25, April 1988.

[Avizienis 1985] A. Avizienis, "The N-version approach to fault-tolerant software", *IEEE Transactions on Software Engineering*, SE-11 (12), pp. 1491-501, 1985.

[Avizienis *et al.* 1987] A. Avizienis, M. R. Lyu and W. Schuetz, *In search of effective diversity: a six-language study of fault-tolerant flight control software*, UCLA Computer Science Department, N°CSD-870060, November 1987 1987.

[Avizienis & Yutao 1999] A. Avizienis and H. Yutao, "Microprocessor entomology: a taxonomy of design faults in COTS microprocessors", in *7-th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, (C. B. Weinstock and J. Rushby, Eds.), (San Jose, California, USA), pp. 3-23, IEEE, 1999.

[Bishop *et al.* 1986] P. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahll and J. Lahti, "PODS - A Project on Diverse Software", *IEEE Transactions on Software Engineering*, SE-12 (9), pp. 929-40, 1986.

[Bishop 1988] P. G. Bishop, "The PODS diversity experiment", in *Software Diversity in Computerized Control Systems* (U. Voges, Ed.), pp. 51-84, Springer-Verlag, 1988.

[Bishop & Pullen 1988] P. G. Bishop and F. D. Pullen, "PODS Revisited - A Study of Software Failure Behaviour", in *18th International Symposium on Fault-Tolerant Computing*, (Tokyo, Japan), pp. 1-8, IEEE Computer Society Press, 1988.

[Bishop & Pullen 1989] P. G. Bishop and F. D. Pullen, "Failure Masking: A Source of Failure Dependency in Multi-version Programs", in *1st IFIP Int. Working Conference on Dependable Computing for Critical Applications (DCCA-1)*, (A. Avizienis and J.-C. Laprie, Eds.), (Santa Barbara, USA), Dependable Computing and Fault-Tolerant Systems Series, 4, pp. 53-73, Springer-Verlag, 1989.

[Blum & Kannan 1989] M. Blum and S. Kannan, "Designing Programs That Check Their Work", in *21st Annual ACM Symposium on Theory of Computing*, (Seattle, Washington), pp. 86-97, 1989.

[Blum *et al.* 1993] M. Blum, M. Luby and R. Rubinfeld, "Self-Testing/Correcting with Applications to Numerical Problems", *Journal of Computer and System Sciences*, 47, pp. 549-95, 1993.

[Blum & Wasserman 1994] M. Blum and H. Wasserman, "Program Result-Checking: A Theory of Testing Meets a Test of Theory", in *35th Annual Symposium on Foundations of Computer Science*, (Santa Fe, New Mexico), pp. 382-92, 1994.

[Brilliant *et al.* 1990] S. S. Brilliant, J. C. Knight and N. G. Leveson, "Analysis of Faults in an N-Version Software Experiment", *IEEE Transactions on Software Engineering*, SE-16 (2), pp. 238-47, February 1990.

[Burlando *et al.* 1992] P. Burlando, L. Gianetto and M. T. Mainini, "Functional Diversity", in *Software Fault Tolerance - Achievements and Assessment Strategies* (M. Kersken and F. Saglietti, Eds.), pp. 49-113, Springer-Verlag, Berlin, 1992.

[Chen & Avizienis 1977] L. Chen and A. Avizienis, "On the Implementation of N-Version Programming for Software Fault Tolerance during Program Execution", in *1st International Computer Software and Applications Conference, COMPSAC 77*, (New York), pp. 149-55, 1977.

[Chillarege 1996] R. Chillarege, "Orthogonal Defect Classification", in *Handbook of Software Reliability Engineering* (M. R. Lyu, Ed.), Computing, pp. 359-400, McGraw-Hill and IEEE Computer Society Press, 1996.

[Di Giandomenico & Strigini 1990] F. Di Giandomenico and L. Strigini, "Adjudicators for Diverse-Redundant Components", in *9th Symposium on Reliable Distributed Systems (SRDS-9)*, (Huntsville, Alabama), pp. 114-23, IEEE, 1990.

[Eckhardt *et al.* 1991] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Transactions on Software Engineering*, 17 (7), pp. 692-702, July 1991.

[Eckhardt & Lee 1985] 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*, SE-11 (12), pp. 1511-7, December 1985.

[Hatton & Roberts 1994] L. Hatton and A. Roberts, "How accurate is scientific software?", *IEEE Transactions on Software Engineering*, 20 (10), pp. 785-97, Oct 1994.

[Huang *et al.* 1995] Y. Huang, C. Kintala, N. Kolettis and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications", in *25th International Symposium on Fault Tolerant Computing (FTCS-25)*, (Pasadena, California, U.S.A.), pp. 381-90, IEEE Computer Society Press, 1995.

[Hughes 1987] R. P. Hughes, "A New Approach to Common Cause Failure", *Reliability Engineering*, 17, pp. 211-36, 1987.

[Kelly & Avizienis 1983] J. P. J. Kelly and A. Avizienis, "A Specification-Oriented Multi-Version Software Experiment", in *13th International Symposium on Fault-Tolerant Computing*, (Milano, Italy), pp. 120-6., 1983.

[Laprie *et al.* 1990] J. C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", *IEEE Computer*, 23 (7), pp. 39-51, 1990.

[Lee & Iyer 1995] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System", *IEEE Transactions on Software Engineering*, 21 (5), pp. 455-67, May 1995.

[Lindner 1998] A. Lindner, "ANSI-C in safety critical applications-lessons learned from software evaluation", in *17th International Conference on Computer Safety, Reliability and Security (SAFECOMP'98)*, (W. Ehrenberger, Ed.), (Heidelberg, Germany), pp. 209-17, Springer-Verlag, 1998.

[Littlewood 1996] B. Littlewood, "The impact of diversity upon common mode failures", *Reliability Engineering and System Safety*, 51, pp. 101-13, 1996.

[Littlewood 2000] B. Littlewood, "The use of proof in diversity arguments", *IEEE Transactions on Software Engineering*, 26 (10), pp. 1022-3, October 2000.

[Littlewood & Miller 1989] B. Littlewood and D. R. Miller, "Conceptual Modelling of Coincident Failures in Multi-Version Software", *IEEE Transactions on Software Engineering*, SE-15 (12), pp. 1596-614, December 1989.

[Littlewood *et al.* 1999] B. Littlewood, P. Popov and L. Strigini, "A note on reliability estimation of functionally diverse systems", *Reliability Engineering and System Safety*, 66, pp. 93-5, 1999.

[Littlewood *et al.* 2001] B. Littlewood, P. Popov and L. Strigini, "Modelling software design diversity - a review", *ACM Computing Surveys*, to appear, 2001.

[Littlewood *et al.* 2000] B. Littlewood, P. Popov, L. Strigini and N. Shryane, "Modelling the effects of combining diverse software fault removal techniques", *IEEE Transactions on Software Engineering*, 26 (12), to appear, December 2000.

[Lyu 1995] M. R. Lyu (Ed.), *Software Fault Tolerance*, Trends in Software, 337p., Wiley, 1995.

[Lyu & He 1993] M. R. Lyu and Y. He, "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm", *IEEE Transactions on Reliability*, R-42 (2), pp. 179-89, June 1993.

[McVittie *et al.* 1992] T. I. McVittie, J. P. J. Kelly and W. I. Yamamoto, "An empirical investigation of the effect of formal specifications on program diversity", in *Second International Working Conference on Dependable Computing for Critical Applications*,

DCCA-2, (J. Meyer and R. Schlichting, Eds.), (Tucson, Arizona, U.S.A.), Dependable Computing and Fault-Tolerance series, 6, (A. Avizienis, H. Kopetz and J. C. Laprie, Eds.), pp. 219-40, Springer-Verlag, 1992.

[Migneault 1982] G. E. Migneault, *The Cost of Software Fault Tolerance*, NASA Langley Research Center, Technical Memorandum, N°TM-84546, September 1982.

[MoD 1996] MoD, *Safety Management Requirements for Defence Systems*, U.K. Ministry of Defence, Defence Standard, N°00-56, Issue 2, December 1996.

[MoD 1997] MoD, *Requirements for Safety Related Software in Defence Equipment*, U.K. Ministry of Defence, Defence Standard, N°00-55, Issue 2, August 1997.

[Neumann 1995] P. G. Neumann, *Computer related risks*, Addison Wesley, 1995.

[Poledna 1994] S. Poledna, "Replica Determinism in Distributed Real-Time Systems: A Brief Survey", *Real-Time Systems Journal*, 6, pp. 289-316, 1994.

[Reason 1990] J. Reason, *Human Error*, Cambridge University Press, 1990.

[Saglietti 1991] F. Saglietti, "A Classification of Software Diversity Degrees Induced by an Analysis of Fault Types to be Tolerated", in *5th International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems. Tests, Diagnosis, Fault Treatment*, (M. Dal Cin and W. Hohl, Eds.), (Nuernberg, Germany), pp. 383-95, Springer-Verlag, 1991.

[Saglietti *et al.* 1992] F. Saglietti, W. Ehrenberger and M. Kersken, *Software-Diversität für Steuerungen mit Sicherheitsverantwortung*, Schriftenreihe der Bundesanstalt für Arbeitsschutz, -Forschung- Fb 664, Dortmund, 1992.

[Shimeall & Leveson 1991] T. J. Shimeall and N. G. Leveson, "An empirical comparison of software fault tolerance and fault elimination", *IEEE Transactions on Software Engineering*, 17, pp. 173-82, 1991.

[Strigini 1990] L. Strigini, *Software Fault Tolerance*, PDCS ESPRIT Basic Research Action, Technical Report, N°23, July 1990.

[Strigini & Avizienis 1985] L. Strigini and A. Avizienis, "Software Fault-Tolerance and Design Diversity: Past Experience and Future Evolution", in *4th IFAC Workshop SAFECOMP'85*, (Como, Italy), pp. 167-72, 1985.

[Voges 1994] U. Voges, "Software diversity", *Reliability Engineering and System Safety*, 43 (2), pp. 103-10, 1994.

[Westerman *et al.* 1995] S. J. Westerman, N. M. Shryane, C. M. Crawshaw, G. R. J. Hockey and C. W. Wyatt-Millington, "Cognitive Diversity: A Structured Approach to Trapping Human Error", in *SAFECOMP'95: 14th International Conference on Computer Safety, Reliability and Security*, (G. Rabe, Ed.), (Belgirate, Italy), pp. 142-55, Springer-Verlag, 1995.

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