Review of Software Design Diversity

P.G. Bishop, Adelard

1 Introduction

Generally speaking, diversity is a protection against *uncertainty*, and the greater the uncertainty (or the greater the consequences of failure), the more diversity is employed. This concept of "defence in depth" is reflected in, for example, aircraft control systems and nuclear plant protection. At the general systems engineering level, diversity is an established approach for addressing critical applications. Such an approach tends to utilise sub-systems that are functionally different (e.g. concrete containment, shutdown rods, boronated water, etc.). Even when subsystems perform a broadly similar function, the implementation technology can differ (e.g. using discrete logic or a computer system).

With the advent of computers, *N*-version software diversity has been proposed [Avi77] as a means of dealing with the uncertainties of design faults in a computer system implementation. Since that time there have been a number of computer systems which are based on the diverse software concept including railway interlocking and train control [And81], Airbus flight controls [Tra88], and reactor protection [Vog82], [Con88].

The main question to ask is "does software diversity buy you more reliability?". From a systems engineering viewpoint, if the software diversity is ineffective or too costly, there may be alternative design options at the systems engineering level that are more cost-effective. The remainder of this paper will attempt to address this question, by summarising research in this area, reviewing the practical application of diversity, and discussing where diversity can be most effectively applied.

2 *N*-version Programming Research

One underlying assumption behind *N*-version programming is that independent developments will produce diverse program faults and this will minimise the likelihood of coincident failures. A much stronger assumption is that "ideal" diverse software would exhibit *failure independence*. In this model the probability of simultaneous failure of a pair of programs A and B is simply $Pf_A.Pf_B$ where *Pf* is the probability of failure for a program execution.

Over the years a range of experiments have been mounted to test these underlying assumptions [Vog94]. The experiments have examined factors that could affect the diversity of the development process, including:

- independent teams
- diverse specification and implementation methods
- management controls

The common features in all these experiments are the use of independent teams to produce diverse programs, followed by acceptance testing the diverse versions, and some form of

comparison testing (either against each other or against a "golden" program) to detect residual faults and estimate reliability. Some typical *N*-version experiments are summarised below:

Experiment	Specs	Languages	Versions	Ref.
Halden, *Reactor Trip*	1	2	2	[Dah79]
NASA, *First Generation*	3	1	18	[Kel83]
KFK, *Reactor Trip*	1	3	3	[Gme80]
NASA/RTI, *Launch Interceptor*	1	3	3	[Dun86]
UCI/UVA, *Launch Interceptor*	1	1	27	[Kni86a]
Halden (PODS), *Reactor Trip*	2	2	3	[Bis86]
UCLA, *Flight Control*	1	6	6	[Avi88]
NASA (2nd Gen.) *Inertial Guidance*	1	1	20	[Eck91]
UI/Rockwell, *Flight Control*	1	1	15	[Lyu93]

Table 1: Some Software Diversity Experiments

The performance of *N*-version programming has been assessed by a number of different criteria, including:

- diversity of faults
- empirical reliability improvement
- comparison with the independence assumption

Some of the main results in these areas will be discussed below.

2.1 Fault diversity

Most of the early research in this area [Dah79], [Gmei80], [Kel83], [Dun86] focused on analysing the software faults produced and the empirical improvement in reliability to be gained from design diversity.

One common outcome of these experiments was that a significant proportion of the faults were similar, and the major cause of these common faults was the specification. Since faults in the design and coding stages tended to be detected earlier, quite a high proportion of specification-related faults were present in the final program versions. For example in the KFK experiment, 12 specification faults were found out of a total of 104, but after acceptance testing 10 specification-related faults were still present out of a total of 18. The major deficiencies in the specifications were incompleteness and ambiguity which caused the programmer to make incorrect (and potentially common) design choices (e.g. in the KFK experiment there were 10 cases where faults were common to two of the three versions).

An extreme example of this general trend can be found in the Project on Diverse Software (PODS) [Bis86]. The project had three diverse teams (in England, Norway and Finland) implementing a simple nuclear reactor protection system application. With good quality control

and experienced programmers *no design-related faults were found* when the diverse programs were tested back-to-back. All the faults were caused by omissions and ambiguities in the requirements specification. However, due to the differences in interpretation between the programmers, five of the faults occurred in a single version only, and two common faults were found in two versions.

Clearly any common faults limit the degree of reliability improvement that is possible, and it would certainly be unreasonable to expect failure independence in such cases. Some of the experiments have incorporated diverse specifications [Kel83], [Bis86] which can potentially reduce specification-related common faults. In general it was found that the use of relatively formal notations was effective in reducing specification-related faults caused by incompleteness and ambiguity. The value of using diverse specifications on a routine basis is less obvious; the performance in minimising common design faults is uncertain, and there is a risk that the specifications will not be equivalent. In practice, only a single good specification method would be used unless it could be shown that diverse specifications were mathematically equivalent.

The impact of the programming language has also been evaluated in various experiments such as [Dah79], [Bis86], [Avi88]. In general fewer faults seem to occur in the strongly typed, highly structured languages such as Modula 2 and Ada while low level assembler has the worst performance. However the choice of language seems to bear little relationship to the incidence of common specification-related or design-related faults, or the eventual performance the programs after acceptance testing.

In recent years there has been a focus on the "design paradigm" for *N*-version programming to improve the overall quality and independence of the diverse developments [Avi88], [Lyu93]. This paradigm requires the identification and specification of the key cross-check points (cc-points) where voting is required between the diverse implementations. Another important feature of the model is the protocol for communication between the development teams and the project co-ordinator. When a problem is identified in the specification, a revision is broadcast to all teams and subsequently followed-up to ensure that the update has been acted upon. This helps to remove specification-related faults at an earlier stage. It is of course important to have a good initial specification since the project co-ordinator can become very overloaded. This situation was observed in the NASA experiment [Kel88] where there was an average of 150 questions per team compared with 11 questions per team in [Lyu93].

The first experiment using this paradigm (the UCLA Six Language Experiment) [Avi88] found only two specification-related common faults (out of a total of 93 faults). These were caused by procedural problems (illegible text and failure to respond to a specification change) and the paradigm was modified to eliminate these problems. In the following experiment [Lyu93], no specification-related faults were found after acceptance testing. The acceptance testing applied was far more extensive than earlier experiments and a lower number of residual faults were found compared with earlier experiments (e.g. around 0.5 faults/KLOC after one acceptance test, 0.05 after both acceptance tests). This is a significant improvement on around 3 faults/KLOC for the Knight and Leveson experiment and 1 fault/KLOC for the NASA experiment.

Perhaps the most important improvement to be noted is the reduction in identical or very similar faults. The following table shows how many similar faults were created during development and includes both common implementation faults and specification faults. The table also shows the *fault span* (the number of versions in which the same fault exists).

Experiment	Similar faults	Max fault span	Versions
Knight and Leveson	8	4	27
NASA	7	5	20
UCLA Flight Control	2	2	6
UI/RI Flight Control	2	2	12

Table 2: Span of similar Faults

One way of interpreting these results is to assess the capability of a set of diverse versions to mask faults completely. This can be done by computing the odds that a randomly chosen tuple of program versions will contain a majority of fault-free versions.

If we assume that, after development, there is a probability *p* of a fault-free program, then it is easy to show that the chance of a fault-masking triple is:

$$p^2(3-2p)$$

So where the chance of a fault-free version is 50%, the chance of creating a fault-masking triple is also 50%. The odds of fault masking triples are computed below for a number of experiments based on the quoted number of fault-free versions.

Experiment	Versions	Fault-free	Fault-masking Triples (%)
Knight and Leveson	27	6	12.6
NASA	20	10	50.0
UI/RI (AT1)	12	7	68.4
UI/RI (AT2)	12	11	98.0

Table 3: Probability of Complete Fault Masking (Triple)

Note that the table computes the probability of selecting a failure-masking triple from an arbitrarily large population of versions. This does not correspond exactly to the number of failure masking triples that can be constructed from the finite set of versions available in the experiments. For example in the UI/RI (AT2) case, 100% of triples are failure-masking.

The above analysis works on the pessimistic assumption that all faulty versions contain similar faults that cannot be masked. If however the odds of faults being dissimilar are incorporated, then the chance of a fault-masking triple is increased. For example in the Knight and Leveson experiment, the maximum fault span is 4 in 27 programs. If we assume the probability of dissimilar or no faults to be 85% (23:27) the odds of selecting a fault-masking triple would increase to 94%.

In the NASA experiment, the largest fault span is 5 (out of 10 faulty versions). Using a figure of 75% for the odds of a version that is fault-free or dissimilar, the odds of failure masking

triple increase to 84%. This estimate is probably an upper bound since dissimilar faults do not guarantee dissimilar failures, so complete masking of failures is not guaranteed.

Similar analyses can be performed for a fail-safe pair (a configuration that is often used in shutdown systems). In this case, a fail-safe action is taken if either channel disagrees, so the pair is only unsafe if both channels are faulty. This would occur with probability $(1-p)^2$, which means that the risk of unsafe pair can be up to 3 times less than the risk of a non-masking triple. Some example figures are shown below.

Prob. version Fault-free	**Prob. of Fault-masking Triple**	**Prob. of Fail-safe Pair**
0.5	0.50	0.75
0.75	0.84	0.94
0.9	0.97	0.99

Table 4: Probability of Fail-safe Pair

The use of odds for such applications is rather problematic, and might not be regarded as an acceptable argument especially where the software is safety-related. A more convincing argument could be made if it could be demonstrated that reliability is improved even when the majority of channels are faulty. The improvement that can actually be achieved is determined by the degree of failure dependency between the diverse channels, and this topic is discussed in the following section.

2.2 Evaluation of failure dependency

One strong assumption that can be made about diversity is that the failures of diverse versions will be independent. An experimental test of the assumption of independence was reported in [Kni86a]. In this experiment, a set of 27 programs were implemented to a common Missile Launch Interceptor specification. This experiment rejected the failure independence assumption to a high confidence level. Furthermore, these dependent failures were claimed to be due to *design faults only*, rather than faults in the specification. Analysis of the faults within the 27 programs showed that programmers tended to make similar mistakes.

Obviously the specification would be a potential source of similar mistakes and hence dependent failures, but the specification was claimed not to affect the outcome. The main justifications for this claim were careful independent review and that fact that it had built on the experience of an earlier experiment [Dun86].

At the time there was some dispute over the results of this experiment, particularly over the realism of an experiment which used students rather than professional software developers. However, the results of the empirical study were supported by a theoretical analysis of coincident failure [Eck85] which showed that, if mistakes were more likely for some specific input values, then dependent failures would be observed. This theory was later refined [Lit89] to show that it was possible to have cases where dependent failures occurred *less frequently* than predicted by the independence assumption. The underlying idea here is that the “degree of difficulty” distribution is not necessarily the same for all implementors. If the distribution can be altered by using different development processes, then failures are likely to occur in different regions of the input space, so the failures could in principle be negatively correlated.

Another experimental evaluation of failure dependency was made in a follow-up to the PODS project [Bis87]. Rather than testing for independence, this experiment measured the *degree of dependency* between the faults in early versions of the PODS programs (which contained both specification- and implementation-related faults).

The experiment used modified versions of the original PODS programs where individual faults could be switched on, so it was possible to measure the individual and coincident failure rates of all possible fault pairs and make comparisons with the independence assumption. For comparison purposes a *dependency factor D* was defined as the ratio of actual coincident failure rate to that predicted by the independence assumption, i.e.:

$$D=P_{ab}/(P_a.P_b)$$

The observed distribution of dependency factors for the fault pairs is summarised in the figure below. In this diagram independent fault pairs would have a dependency factor of unity. In practice, the distribution of dependency factors ranged from strong positive correlation to strong negative correlation. These extreme values were well beyond the calculated 95% confidence limits for the independence assumption.

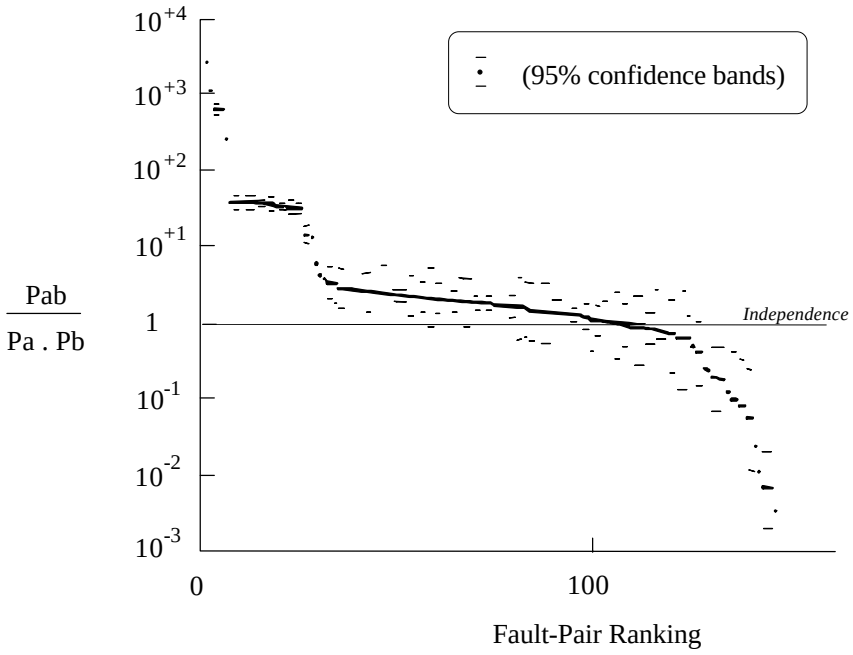

Figure 1: Dependency Factors for PODS Fault Pairs

On analysis, it was found that the strongly negatively correlated failures occurred between similar functions in the two diverse programs, however the faults exhibited a *failure bias* on a binary value. In one program the failure (when it occurred) produced a "1", while in the other program, the failure value was always "0". This meant that coincident failures would never occur because one function was always correct when the diverse function was faulty. Such biased failures are an interesting example of a specific mechanism which could result in negatively correlated failures.

Most of the positively correlated (high dependency factor) fault pairs were related to the known common mode faults caused by problems in the requirements specification. However some strange clusters of high dependency fault pairs were observed which could not be accounted for by the known common mode faults (see the high dependency "plateaus" in the figure above).

On examination, two of the high dependency plateaus were associated with failures on the same single-bit output, but it was difficult to explain the very similar levels of dependency, as there was little or no commonality between the input data variables, the faults, or the observed failure rates.

Eventually it was found that these common dependency levels were caused by "error masking" [Bis89]. The single-bit output value was determined by an "OR" of several logical conditions as shown in the figure below. This meant that an incorrectly calculated condition value could not be observed at the output unless all the other logical conditions were zero. This masking of internal errors causes dependent failures to be observed *even if the internal error rates are independent*.

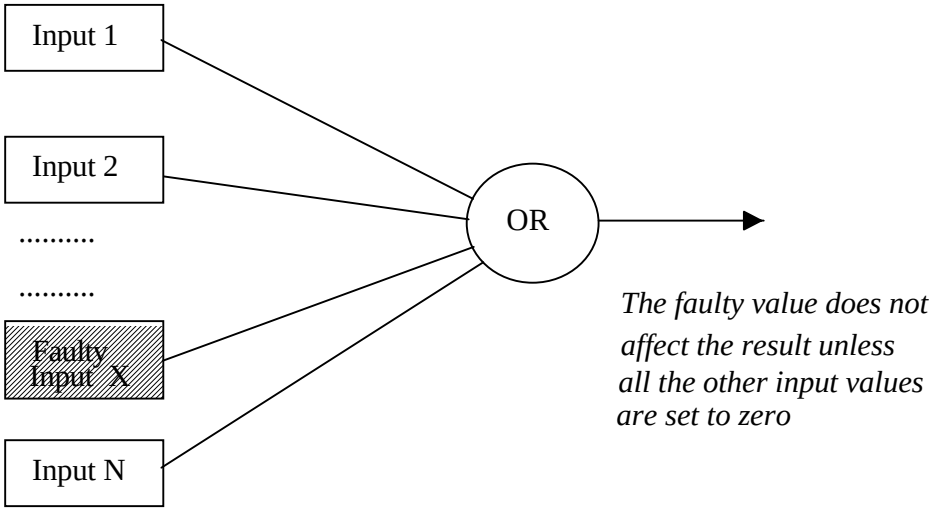

Figure 2: Error-masking with OR logic

The OR gate tends acts as a common "shutter" for the internal errors and will tend to open simultaneously on both versions because it is controlled by common program input values. This "shutter effect" results in correlation of the externally observed failures; high internal error rates have low externally observable failure rates (due to shutter closure), but exhibit high coincident failure rates (when the shutter opens).

In the case of the OR gate, it can be shown that for independent internal failures, the dependency factor approximates to $1/P0$, where $P0$ is the probability of observing a zero (logical false) at the OR output in a fault-free program. This dependency level is not affected by the error rates of the internal faults, which explains why the plateau of similar dependency factors were observed.

This error masking effect is a general property of programs. Any output variable whose computation relies on masking functions is likely to exhibit dependent failures in diverse implementations. Simple examples of masking functions are AND gates, OR gates, MAX and MIN functions or selection functions (IF.. THEN.. ELSE, CASE, etc.). In all these functions it is possible to identify cases where a faulty input value can be masked by the remaining inputs to yield a correct result. In fact any N to M mapping function is capable of some degree of masking provided M is smaller, so this phenomenon will occur to some extent in all programs.

Interestingly, the Launch Interceptor example contains a large number of OR gates within its specification (see the figure below). The failure masking process for the Launch Interceptor Conditions (LICs) is somewhat different from the PODS example, but analysis indicates that

dependency factors of one or two orders of magnitude are possible [Bis91]. In experimental terms therefore, the Launch Interceptor example could be something of a “worst case” in terms of the degree of error masking that might be anticipated.

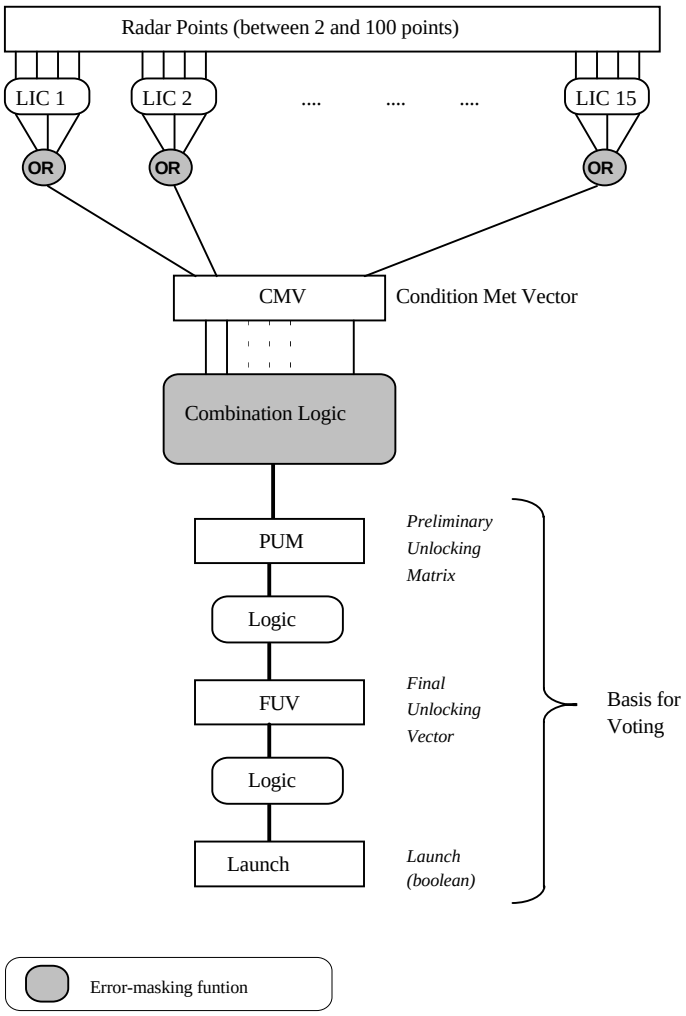

Figure 3: Error Masking in the Launch Interceptor Application

The more recent experiments by NASA, UCLA and UI/RI have also attempted to estimate reliability improvement. In these examples, there is very little scope for error masking. For example the RSDIMU inertial measurement system [Kel88] has a limited number of inputs and performs arithmetic computations which have limited potential for masking. This is also true for the aircraft automatic lander example [Avi88]. If error masking were the only source of dependency, low coincident failure rates might be expected. In practice, the reliability improvements for arbitrary triples were still limited by dependent failures.

The NASA program versions contained significant numbers of near-identical design and specification-related faults (largely caused by misunderstanding of the specification or by a lack of understanding of the problem). This affected the achieved reliability very markedly and a statistically unbiased estimation of the reliability showed that the average failure rate of a triple was only 4 times lower than a single version.

However the variability in improvement is very wide. Depending on the operating mode, between 60% to 80% of triples successfully masked *all* failures. This is not too surprising

since 50% of the triples contain only one faulty version (as discussed in the previous section). Of the remaining triples, some provided reliability improvements of possibly one order of magnitude, but the others (containing a high proportion of similar faults) exhibited very modest improvements (e.g. 10% of the triples gave a 20% reduction or less).

While common faults were the major cause of poor performance, it is interesting to note that two dissimilar faults lead to high levels of coincident failure. This may again be an inherent feature of the program function (but unrelated the error masking effect). In the NASA experiment there is one function called the Fault Detection and Isolation (FDI) module. This module is responsible for detecting faulty sensors so that subsequent modules can compute the acceleration using the remaining good sensors. The two dissimilar faults affected this module. A fault in this module should cause too many (or too few) sensors to be used. In either circumstance the computed result will disagree with the "golden" value. The significant feature is that failures in the faulty versions are likely to coincide with a change in the failure status of a sensor. This would cause a burst of coincident failures (even if the reasons for the sensor diagnosis errors differ).

The reliability improvements achieved in the UI/RI flight controller example are somewhat better. Using the versions obtained after acceptance test 1 (AT1), the reliability improvement for an average triple was around 13. This estimate did not include the error correction capability of the voting system. If this is included the average reliability improvement factor is increased to around 58.

The improved performance is largely the result of better quality programs containing fewer common faults which spanned fewer program versions. While there is no direct analysis of the distribution of reliability improvements over the triples, there is likely to be a significant amount of variability. It is known that at least 68% of triples must mask all failures (from the earlier fault distribution analysis), while the triples with similar faults (under 10%) are likely to exhibit only modest improvements. The remaining triples might exhibit quite large reliability improvements. In principle the improvement might approach that predicted by the independence assumption (as they did in PODS), but other sources of dependency such as the mechanism discussed in the NASA experiment could limit the gains.

The main lesson to be learned from these experiments is that the performance of *N*-version programming is seriously limited if common faults are likely. The most probable sources are likely to be common implementation mistakes (e.g. omitted cases or simple misunderstanding of the problem) together with omissions and ambiguities in the specification. These factors need to be closely controlled to minimise the risks and the diverse programming paradigm seems to have made an important contribution. The risks could be further reduced if it possible to predict where such common faults are likely to occur so that more effort can be devoted to these areas. The use of metrics to identify "tough spots" in the program functions [Lyu94] seems to be a promising approach.

3 Practical Application of *N*-version Programming

In addition to the research work undertaken to assess the potential for reliability improvement, it is also necessary to examine other issues that could affect the decision to adopt *N*-version programming. Some of the key areas are examined below.

3.1 Maintaining consistency

N-version programming can impose additional design constraints in order to deal with the diversity in computation. The problems of dealing with such differences in computed values

were noted in [Bri89]. The main difficulty is that, within some given precision, there can be multiple correct answers—any of the answers would be acceptable but to maintain consistency, one should be chosen. This is particularly important where the program reaches a decision point where alternative but incompatible actions may be taken. A simple example of this is a railway interlocking system—two equally acceptable alternatives are to stop one train and allow the other to pass or vice versa, but the choice must be agreed. Furthermore when there are a number of related outputs (such as a set of train interlock outputs) it is not sufficient to take a vote on individual values—a consistent set of values must be selected. Without such arrangements it is, in principle, possible for all the diverse programs to make valid but divergent choices. In order to achieve consistency, diverse programs must implement cc-points to perform a vote at all key decision points. This design constraint is not unique to diverse designs. Quite often redundant channels with *common* software require cc-points to ensure consistency of action (e.g. for multiple sensor readings which can have slightly different values). The main difficulty with cc-points in diverse programs is that they impose additional design constraints which could compromise the diversity of the software designs.

The above discussion shows that design constraints have to be imposed in order to implement consistent comparisons when decisions are based on numerically imprecise data. It is therefore easiest to apply diversity in applications where such indeterminacy either does not exist or does not affect any key decisions. Fortunately quite a few safety-related applications fall into this category. For example, railway interlocking depends on binary status values (e.g. signal on or off) which are not indeterminate. Reactor protection systems may base their decisions on numerical calculations, but internal consistency is not needed to maintain safety. In borderline conditions, either a trip or no-trip action would be acceptable, and the only effect of small discrepancies would be to introduce a small delay until reactor conditions changed. The same argument applies to the launch interceptor application; in practice the discrepancies would only delay a launch by a few execution cycles since fresh data are arriving continuously. Many continuous control applications could, in principle, operate without cc-points. This is possible because most control algorithms are self-stabilising and approximate consistency can be maintained by the feedback of the data from the plant.

At present, most of the industrial applications of design diversity fall into the class where consistent comparisons are not a major issue. For example the rail interlocking system [Hag87] operates on binary data, while the Airbus controller [Tra87] is an example of continuous control where consistency is checked within some window. Actually the Airbus design utilises the concept of a diverse self-checking pair—one diverse computer monitors the other and shutdown occurs if the diverse pair disagrees outside some window. On shut-down the functions are taken over by other self-checking pairs (with a different mix of software functions). The CANDU reactor protection system utilises two diverse computer systems that trigger separate shutdown mechanisms [Con88] so no voting is required at all between the diverse systems.

In more complex systems, it is undeniable that standardised intermediate voting points would have to be identified in order to prevent divergent results. The voting schemes used in such cases are very similar to those that would be required for a parallel redundant system and would be essential in any case in order to maintain system availability. It is not clear how much these voting constraints compromise design diversity—most of the observed faults in diversity experiments seem to occur in specific functional modules and are related to the programmer’s knowledge or specification difficulties.

3.2 Testing

One potentially cost-saving feature of diversity is the capability for utilising back-to-back testing to reduce the effort in test design. Experiments have shown that this form of testing is as effective as alternative methods of fault detection; in a re-run of PODS using back-to-back testing, 31 out of 33 unique faults were found [Bis87]. The remaining 2 faults were found by back-to-back testing with final "silver" programs against the original "perfect" PODS programs. Of these two faults one was a minor computation error which was within the specified tolerance, and the other might not be a fault at all (given a quite reasonable interpretation of the specification). The cross-check also revealed a common fault in the original "perfect" programs caused by the use of an incorrect acceptance test.

Back-to-back tests on the Launch Interceptor programs were shown to detect the same number of faults (18 out of 27) as alternative, more labour-intensive, methods [Lev90]. An earlier experiment on a different program example [Lev88] also found that back-to-back testing was as good as any alternative method at detecting faults (123 out of 270). The lower detection performance of these two experiments compared with the PODS re-run might be partly explained by a difference in procedure; in the PODS re-run, any discrepancy resulted in an analysis of *all three programs*, while in the other two experiments fault detection was defined to require a successful majority vote. In this context a "vote" assumes that if one program disagrees with the other two and the outvoted program is actually correct, the fault is unrevealed. The use of voting rather than a straight comparison as the detection test can therefore leave two-fold common faults or three-fold distinct faults undetected. Another procedural difference was that the PODS programs were re tested after fault removal, which helps to reveal faults masked by the prior faults so the fault detection efficiency progressively improves.

One of the problems encountered in such testing is deciding when a true failure has occurred; as noted earlier there can be small differences in computation which result in discrepancy. Are these discrepancies really faults? In actual real-time operation, such minor differences are likely to be resolved soon afterwards. So, viewed in relation to the real-world need, the system performs its function. Conventional engineering uses the concept of acceptable "tolerances", and perhaps it would be fruitful if such concepts could be used more widely in software engineering.

4 Discussion and Conclusions

Almost from the beginning, it was recognised that *N*-version programming was vulnerable to common faults. The primary source of common faults arose from ambiguities and omissions in the specification. Provided the difficulties with the specification were resolved, it was hoped that the implementation faults would be dissimilar and exhibit a low level of coincident failure (possibly approaching the levels predicted under the independence assumption). The Knight and Leveson experiment did a service to the computing community as a whole by showing that failure independence of design faults cannot be assumed.

This result is backed up by later experiments and qualitative evidence from a number of sources which shows that common design mistakes can be made. Also, from a theoretical standpoint, it has been shown that any variation in the degree of difficulty for particular input values will result in failure dependency. Furthermore the error masking phenomenon has been identified as an explicit mechanism that will cause failure dependency.

This may paint too gloomy a picture of the potential for *N*-version programming, because:

- Back-to-back testing can certainly help to eliminate design faults, especially if some of the intermediate calculations can be exposed for cross-checking.

- The problems of failure dependency only arise if a majority of versions are faulty. If good quality controls are introduced the risk of this happening can be reduced significantly.

This latter point is supported by the results of recent experiments indicating that the chance of a fault-free version can range from 60% to 90%. In addition, good quality conventional development processes can deliver software with fault densities between 1 and 0.1 faults/KLOC. In this scenario, with small applications, the probability of having multiple design faults can be quite low, and so two fault-free diverse programs will have no problem out-voting the single faulty one.

This is essentially a gambling argument—and it might be reasonable to ask whether you would bet your life on the outcome; however you can only compare this option with the alternative—betting your life on a single program. The relative risks are identical with a 50% probability of a fault-free version and the residual risk only moves slowly in favour of a diverse triple as the probability increases (3 times less risk at 90%, 30 times less at 99%).

This argument presupposes that the chance of a fault-free program would be the same in either case. However *N*-version programming could improve the probability through extensive back-to-back testing, while a single version could benefit if the same resources were devoted to greater testing and analysis. The main argument in favour of *N*-version programming is that it can also give protection against dissimilar faults, so absolute fault-freeness is not required in any one channel—just dissimilarity. This is a less stringent requirement which should have a marked effect on the odds of obtaining an (almost) failure-free system.

Based on its potential for intensive testing and the capability for outvoting dissimilar faults, *N*-version programming can be a useful safeguard against residual design faults, but the main concern is whether software diversity is justifiable in the overall system context. It could be argued that the key limitation to reliability is the *requirements specification*. *N*-version programming can have benefits in this area because the specification is subjected to more extensive independent scrutiny; however this is only likely to reveal inconsistencies, ambiguities and omissions. The system as a whole would still be vulnerable if the wrong requirement was specified. Excessive reliance on software diversity may be a case of diminishing returns (i.e. high conformity to the wrong specification). So it might make more sense to utilise diversity at a higher level (e.g. a functionally diverse protective system) so that there is some defence-in-depth against faults in the requirements.

Nevertheless in systems where there are no real options for functional diversity, and the requirements are well-established (e.g. railway interlocking), then *N*-version programming seems to be a viable option. Perhaps paradoxically, the greatest confidence can be gained from the use of diversity when there is only a low probability of design faults being present in the software. So software diversity is not an alternative to careful quality control—both methods are necessary to achieve high levels of reliability.

5 References

[And81] H. Anderson and G. Hagelin, “Computer Controlled Interlocking System”, *Ericsson Review*, No. 2, 1981

[Avi77] A. Avizienis and L. Chen, “On the Implementation of N-version Programming for Software Fault Tolerance during Execution”, Proc. *the First IEEE-CS International Computer Software and Applications Conference* (COMPSAC 77*)*, Chicago, Nov 1977

[Avi88] A. Avizienis, M.R. Lyu and W. Schutz, “In Search of Effective Diversity: A Six Language Study of Fault Tolerant Flight Control Software”, in *Eighteenth International Symposium on Fault Tolerant Computing* (FTCS 18), Tokyo, June 1988

[Bis86] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll and J. Lahti, “PODS—A Project on Diverse Software”, *IEEE Trans. Software Engineering*, Vol SE-12, No 9, 1986

[Bis87] P.G. Bishop, D.G. Esp, F.D. Pullen, M. Barnes, P. Humphreys, G. Dahll, B. Bjarland, H. Valisuo, “STEM—Project on Software Test and Evaluation Methods”, *Proc. Safety and Reliability Society Symposium* (SARSS 87), Manchester, Elsevier Applied Science, Nov. 1987

[Bis89] P.G. Bishop and F.D. Pullen, “Error Masking: A Source of Dependency in Multi-version Programs”, in *Dependable Computing for Critical Computing Applications*, Santa Barbara, August 1989

[Bis91] P.G. Bishop and F.D. Pullen, “Error Masking: A Source of Dependency in Multi-version Programs”, Paper in: *Dependable Computing and Fault Tolerant Systems*, Vol 4. “Dependable Computing for Critical Applications”, (ed. A. Avizienis and J.C. Laprie), Springer Verlag, Wien-New York, 3-211-822249-6, 1991

[Bri89] S.S. Brilliant, J.C. Knight and N.G. Leveson, “The Consistent Comparison Problem”, *IEEE Trans. Software Engineering*, Vol. 15, Nov, 1989

[Bri90] S.S. Brilliant, J.C. Knight and N.G. Leveson, “Analysis of Faults in an N-Version Software Experiment”, *IEEE Trans. Software Engineering*, Vol. 16, No. 2, Feb, 1990

[Con88] A.E. Condor and G.J. Hinton, “Fault tolerant and fail-safe design of CANDU computerised shutdown systems”, *IAEA Specialist Meeting on Microprocessors important to the Safety of Nuclear Power Plants*, London, May 1988

[Dah79] G. Dahll and J. Lahti, “An investigation into the methods of production and verification of highly reliable software”, *Proc. SAFECOMP 79*

[Dun86] J.R. Dunham, “Experiments in software reliability: life critical applications”, *IEEE Trans. Software Engineering*, Vol SE-12, No. 1, 1986

[Eck85] D.E. Eckhardt and L.D. Lee, “A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Failures”, *IEEE Trans. Software Engineering*, Vol. SE-11, No. 12, 1985

[Eck91] 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 Trans. Software Eng.*, vol. SE-17, no. 7, pp. 692–702, 1991

[Gme80] L. Gmeiner and U. Voges, "Software diversity in reactor protection systems: an experiment", in *Safety of computer control systems*, R. Lauber, Ed., New York: Pergamon, 1980

[Gme88] G. Gmeiner and U. Voges, "Use of Diversity in Experimental Reactor Safety Systems", in *Software Diversity in Computerised Control Systems*, U. Voges, Ed., Springer Verlag, 1988

[Hag87] G. Hagelin, "Ericsson System for Safety Control", *Software Diversity in Computerised Control Systems*, U. Voges, Ed., Springer Verlag, 1988

[Kel83] J.P.J. Kelly and A. Avizienis, "A specification-oriented multi-version software experiment", in *Thirteenth International Symposium on Fault Tolerant Computing* (FTCS 13), Milan, June 1983

[Kel88] J.P.J. Kelly, D.E. Eckhardt, M.A. Vouk, D.F. McAllister, A. Caglayan, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", in *Eighteenth International Symposium on Fault Tolerant Computing (FTCS 18)*, Tokyo, June 1988

[Kni86a] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming", *Proc IEEE Trans on Software Engineering*, Vol SE-12 Jan 1986

[Kni86b] J.C. Knight and N.G. Leveson, "An Empirical Study of the Failure Probabilities in Multi-Version Software", *Proc. FTCS 16*, Vienna, July 1986

[Lev88] N.G. Leveson and T.J. Shimeall, "An Empirical Exploration of Five Software Fault Detection Methods", *IFAC SAFECOMP 88*, Fulda Germany, Nov. 1988

[Lev90] N.G. Leveson, S.S. Cha, J.C. Knight and T.J. Shimeall, "The use of Self Checks and Voting in Software Error Detection: An Empirical Study", *IEEE Trans. Software Engineering*, Vol. 16, No. 4, April 1990

[Lit89] B. Littlewood and D. Miller, "Conceptual modelling of coincident failures in multi-version software", *IEEE Trans on Software Engineering*, vol. SE-15, no. 12, pp.1596–1614, 1989.

[Lyu92] M.R. Lyu, "Software reliability measurements in an n-version software execution environment", *Proc. Int'l Symp. Software Reliability Engineering*, Oct 1992

[Lyu93] M.R. Lyu and Y. He, "Improving the N-Version Programming Process Through the Evolution of a Design Paradigm", *Proc. IEEE Trans. Reliability*, Vol. 42, No. 2, June 1993

[Lyu94] M.R. Lyu, J-H. Chen, A. Avizienis, "Experience in Metrics and Measurements for N-Version Programming", *Int. J. of Reliability, Quality and Safety Engineering*, Vol. 1, No. 1, 1994

[Tra87] P. Traverse, “Airbus and ATR System Architecture and Specification”, *Software Diversity in Computerised Control Systems*, U. Voges, Ed., Springer Verlag, 1988

[Vog82] U. Voges, F. French and L. Gmeiner, “Use of Microprocessors in a Safety-oriented Reactor Shutdown System”, *EUROCON*, Lyngby, Denmark, June 1982

[Vog94] U. Voges, “Software Diversity”, *Reliability Engineering and System Safety*, No. 43, 1994