Contents

9

Software Fault Tolerance by Design Diversity

PETER BISHOP
Adelard, England

ABSTRACT

This chapter reviews the use of software diversity, and especially multi-version programming, as a means of improving system reliability and safety. It considers the theoretical and experimental research undertaken in this field together with some of the more pragmatic issues in implementing diversity. Some alternatives to multi-version programming are also considered. The author concludes that software diversity is a viable option for improving reliability, and the extra costs could well be justifiable in an overall project context. However it is not the only option, and better results might be obtained by employing diversity at a higher level.

9.1 INTRODUCTION

Diversity as a basic concept has been around for many years as reflected in the old adage of "Do not put all your eggs in one basket". A more concrete example of this approach can be given in the Manhattan Project when the first nuclear pile was being constructed. In this case, the diverse shut-down system was a man with an axe who could cut a rope to release the shutdown rods.

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 'defense 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 utilize sub-systems that are functionally different (e.g. concrete containment, shutdown rods, boronated water,

Software Fault Tolerance, Edited by Lyu

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 protection of the Darlington nuclear reactor [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 chapter will attempt to address this question, by summarizing research in this area, reviewing the practical application of diversity, and discussing where diversity can be most effectively applied.

9.2 N-VERSION PROGRAMMING RESEARCH

The key element of N-version programming approach is *diversity*. By attempting to make the development processes diverse it is hoped that the versions will contain diverse faults. It is assumed that such diverse faults will minimize 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. 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. Table 9.1 summarizes some typical N-version 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:

9.2.1 Fault Diversity

Most of the early research in this area [Dah79, Gme80, Vog82, Kel83, Dun86] focused on analyzing the software faults produced and the empirical improvement in reliability to be gained from design diversity.

Table 9.1 Summary of some N-version programming experiments

Experiment	Specs	Languages	Versions	Reference
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]

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 minimizing 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 is discussed in detail in Chapter 2, but one 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 problem. In the following experiment [Lyu92, 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. Table 9.2 shows how many similar faults were created during program development of some experiments, including both common implementation faults and specification faults. Table 9.2 also shows the *fault span* (the number of versions in which the same fault exists).

Table 9.2 Distribution of similar faults in some experiments

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

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 (in a gambling sense) 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 in Table 9.3 for a number of experiments based on the quoted number of fault-free versions.

Note that Table 9.3 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.

Table 9.3 Computed odds of fault masking triples

Experiment	Versions	Fault-free	Fault-masking triple (%)
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

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 version disagrees, so the pair is only unsafe if both versions 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 in Table 9.4:

Table 9.4 Fail-safe probability analysis

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

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 versions are faulty. The improvement that can actually be achieved is determined by the degree of failure dependency between the diverse versions, and this topic is discussed in the following section.

9.2.2 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, Kni86b]. 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 specification was claimed to 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 summarized in Figure 9.1. 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.

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 Figure 9.1). 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 condi-

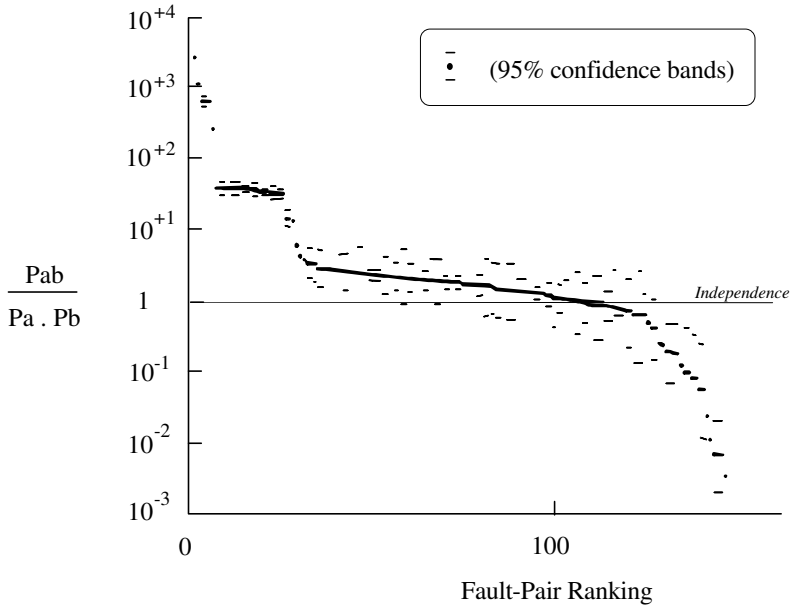

Figure 9.1 Dependency factors for PODS fault pairs

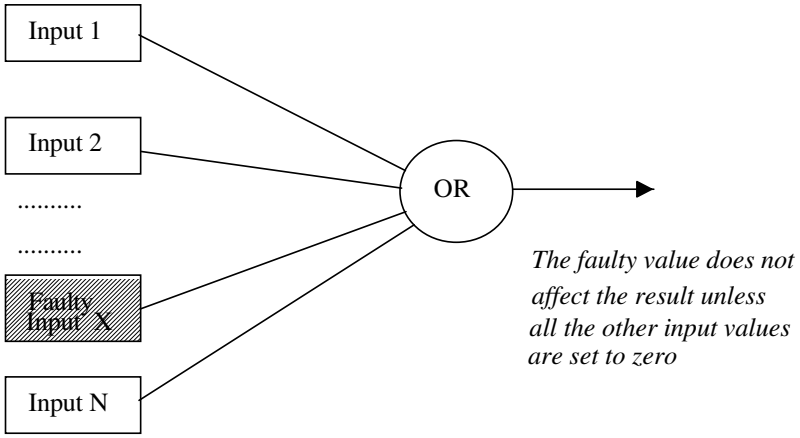

Figure 9.2 Error masking with OR logic

tions as shown in Figure 9.2. 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*.

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 Figure 9.3). 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.

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

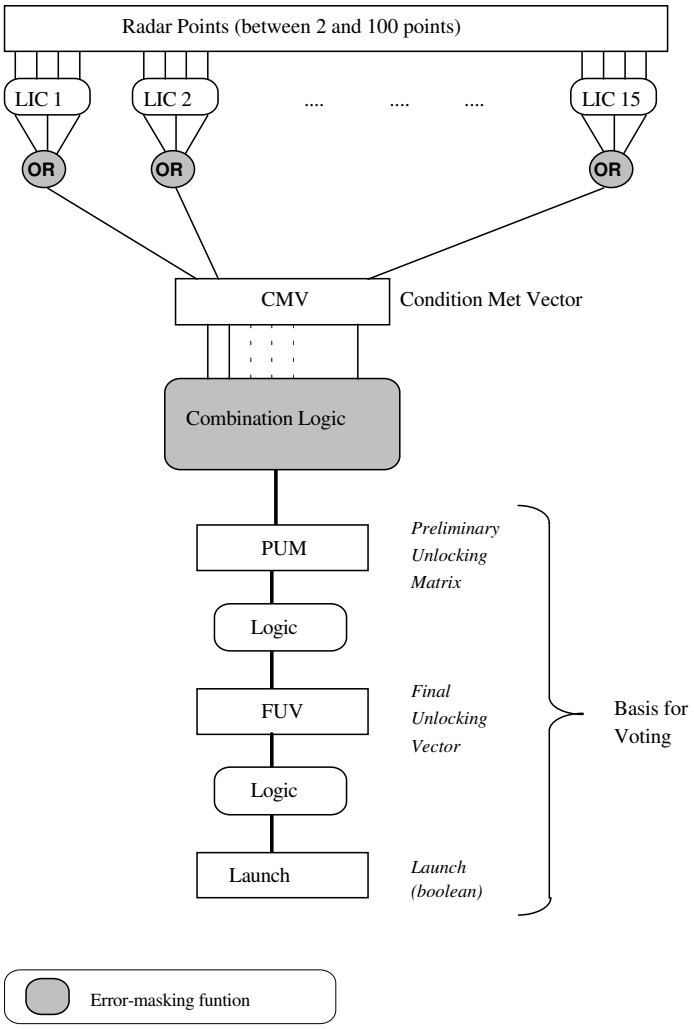

Figure 9.3 The Launch Interceptor application

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 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 minimize 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.

9.3 PITFALLS IN THE DESIGN OF N-VERSION EXPERIMENTS

When N-version programming experiments are performed it is quite easy to introduce common mode failures as an inherent feature of the experimental design. It is important that the experimenter should be aware of these risks. This section gives some examples of experimental design problems which are partly based on the author's own experience on the PODS project.

9.3.1 Faulty Acceptance Tests and Golden Programs

In diversity experiments, the individual programs are normally subjected to acceptance tests before being checked for failure dependency. In subsequent testing, failures are normally established by comparing the results produced by the diverse version against those from a 'golden' program. In the STEM project [Bis87] (a follow-up to PODS) early versions of the PODS programs were tested back-to-back *without an acceptance test*. Surprisingly, the eventual 'silver' programs produced by this process differed from the original 'perfect' PODS programs. On analysis it was found that the common acceptance test used in PODS was faulty (i.e. there was a misinterpretation of the requirements when defining the test data). All the PODS teams 'corrected' their programs to meet the acceptance test, and hence introduced two common mode faults.

On reflection this common mode mechanism is fairly obvious; the acceptance test is essentially just another diverse program which might fail to implement the specification correctly. Excessive reliance on this single 'program' would obviously compromise diversity.

This might have implications for other experiments. In quite a few experimental designs, the acceptance test data is generated by a 'golden' program. In this case, if a fault is activated while generating the acceptance test data, the programs are likely to be modified to mirror this fault (which would typically be an alternative interpretation of the specification). If the golden program is used in subsequent operational testing such faults will be undetectable. This detracts from the experiment, but the remaining population of observable faults should

be realistic examples (even if they are reduced in number). The more interesting case is where a golden program contains a fault but *does not* reflect this in the acceptance test data. In subsequent operational tests it would be possible to observe an apparent coincident failure of *all* the diverse programs. In reality, it is the golden program that has failed, but the apparent effect is a fault that spans all program versions. Fortunately, this situation should be relatively easy to spot and, to my knowledge, this effect has not been reported in the literature.

The only exception to these two extreme scenarios can be found in the Knight and Leveson experiment. The experiment used 200 *randomly generated* test data sets as the acceptance tests for the 27 programs. The risk here is that, since the test data is different for each program version, a fault in golden program could be reflected in some test sets but not in the others. Any program version using a test set that did not reflect the golden program fault would subsequently be deemed to contain the same common fault.

Of course this is not a problem if the golden program and the specification are consistent, but there is some evidence that they are not. The major source of dependent failures in the Launch Interceptor experiment is related to one particular function (embodied in Launch Interceptor conditions 3 and 10). In the published requirements specification, the tests for LICs 3 and 10 are *directional* — testing whether a third point represents a change of direction (or turn) of greater than a specified angle from the direction specified by the first two points [Kni86a]. This would certainly make sense in testing for valid missile tracks where very sharp turns would be unlikely. However, the description of the test 'failure' in the main body of the paper talks in terms of *collinearity* where the three points have to be in approximately a straight line [Kni86a]. Such a definition would permit a second case where the missile performs a 180 degree turn. The 'faulty' programs omitted the 180 degree turn case.

This discrepancy could be explained if the golden program implemented a collinearity test while the requirement specified an 'angle of turn' test. In a subsequent analysis of the faults [Bri90], only three of the 27 programs seem to contain this particular 'omission' (Programs 3, 8 and 25), but around 10 of the programs contain faults associated with incorrect implementations of the 'omitted' case.

Under an hypothesis of 'induced common mode', 24 of the 27 programs were 'corrected' to agree with the golden program, leaving 3 which attempted to implement the specified behavior for LICs 3 and 10. This particular scenario would possible if the probability of observing a 'missing case' failure was around 0.01 per test case. This is quite credible if 90% to 95% of the internal errors in LICs 3 and 10 are masked.

In addition, the need for all 27 programs to be 'corrected', either at the acceptance test stage or in subsequent testing, could result in a high proportion of faults being found in LICs 3 and 10 simply due to incorrect modifications. This is particularly likely if the programmer has to infer the change from the test failures rather than an explicit specification.

It is impossible to be sure whether this particular problem actually occurred in the Knight and Leveson experiment, but it does serve to illustrate that quite subtle sources of induced common mode can exist in an apparently straightforward experimental design.

9.3.2 Diversity and Voting

Many experimenters have noted 'discrepancies' where diverse software disagrees but there is no apparent fault. The discrepancies generally arise because there are differences in computational precision and the choice of algorithm. Discrepancies were certainly noted in the PODS experiment. Fortunately they were quite rare so it was possible to perform a manual

analysis to determine whether the discrepancy was symptomatic of a genuine fault. The discrepancies were infrequent because the comparison scheme included a tolerance band so that differences in the computation of analogue values could be accepted. However, this was not a complete answer because the analogue values had to be compared with thresholds in order to determine to value of certain binary outputs (such as the trip signal). It is therefore possible for one program to compute a value above the threshold and another program to be below the threshold and hence disagree. This event was infrequent in PODS (a few per million) because there were only 3 or 4 comparisons of this type and the 'error band' was small because the calculations were straightforward. Using the design paradigm discussed in chapter 2, such key decision points (cc-points) should be identified from the functional specification and be voted upon (e.g. so that a single voted value is compared with the threshold).

If an experiment permits inconsistent comparisons, this can be a source of apparent common mode failure (especially in large scale-experiments). Let us suppose that a value (as computed by the golden program) is close to a threshold. On average around half the programs would compute values on the other side of the threshold *and would be deemed to fail simultaneously*. So even if all the programs are 'correct' in the sense that they conform to the specification with adequate precision, the experiment would observe a high proportion of coincident failures.

Other experiments (such as the aircraft landing example) explicitly include cc-points and voting within tolerance bands so that this problem is overcome. In such examples, consistency is very important because data from past cycles is reused and so the effect of an inconsistent comparison (and the associated dependent failures) can persist for many subsequent executions cycles.

Like PODS, the Knight and Leveson example does not contain cc-points. However the Launch Interceptor specification requires a large number of comparisons — up to 1500 values have to be computed and compared against thresholds on every execution cycle. There was an attempt to ensure consistency by requiring all programmers to use single precision floating point and a REALCOMPARE function that limited the accuracy of all comparisons (including those used in algorithms) to 6 decimal digits. This requirement does not improve comparison consistency, and can increase the size of the 'error band' for the computed values (especially where algorithms make heavy use of the REALCOMPARE function).

It is not obvious what effect inconsistent comparisons had on the overall experiment. It is notable that in a subsequent analysis of the faults [Bri90], at least 13 of the 45 faults are associated with accuracy problems caused by using different algorithms. Under a different voting scheme these might not have been regarded as faults, so the failure and the associated failure dependency would have been removed.

9.4 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.

9.4.1 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 voting arrangements before all key decision points. This enforced commonality could compromise the diversity of the individual 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 is arriving continuously. Many continuous control applications could, in principle, operate without cc-points. This is possible because most control algorithms are self-stablizing 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 [Bri93] is an example of continuous control where consistency is checked within some window. Actually the Airbus design utilizes 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 utilizes 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 standardized 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 programmers knowledge or specification difficulties.

9.4.2 Testing

One potentially cost-saving feature of diversity is the capability for utilizing 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 'golden' 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 (e.g. if the nuclear power level increased a bit more, or another radar echo is received). 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 software engineering.

9.4.3 Cost of Implementation

In general, the development and maintenance costs for three-fold diversity could be twice that of a single development and less than double for two-fold diversity [Bis86, Hag87, Vog94]. The increases are not directly proportional to the number of diverse versions because some savings can be made on shared activities (such as requirements specification and the provision of test environments).

If the use of diversity doubled the cost of the whole project, then this could well be an impediment to the practical application of design diversity. However in making such choices we have to examine the overall cost implications to the project:

1. For some applications, only a small part of the functionality is safety critical. For example in the Ericsson railway interlocking system [Hag87], the majority of the system is concerned with quite complex computations to make train control decisions. The control

commands are submitted to the interlocking software and it is only the interlocking software which is diversely implemented.

2. Some industries are subject to official regulation (e.g. aircraft and nuclear power). In these industries, the costs of *demonstrating safety* can far outweigh the development costs. For example the Sizewell 'B' nuclear power station computer-based protection system has absorbed over 500 man-years of effort in safety assessment [Bet92, War93], and this does not include any lost income associated with licensing delays. In such circumstances, diversity (although not necessarily software diversity) has been used to make a more convincing safety case.
3. Acceptable alternatives to diversity may also increase the cost. For example, formal methods can be used to prove that a program meets its specification. This might give a high confidence that there are no software design faults. However such development approaches require highly specialized and expensive staff, and there are still residual risks of faults in the proof process, the compiler and the underlying hardware.

9.5 ALTERNATIVES TO N-VERSION PROGRAMMING

This chapter has concentrated on N-version programming, but the diversity concept can be utilized in others ways. The well-known recovery block method [Ran75] is excluded from this discussion since it has been described in detail in Chapter 1.

One interesting idea is the comparison of data derived from an *executable specification* with the behavior of the final implemented program [Blo86]. In this case the specification is a mathematically formal description which can be executed (albeit rather slowly) to generate test cases which can be compared with the actual behavior of the implemented program. This has the advantages of back-to-back testing, without the necessity for a full implementation of a second diverse version.

Another highly novel approach to diversity has been suggested in recent years, and this is *data diversity* [Amm88]. The basic idea here is that the *same program* can be utilized, but run several times with different input data, and then the results are combined using a voter. Part of the motivation behind this approach is the observation that faults tend to manifest themselves as contiguous areas in the input space, and this appears to be a general phenomenon [Fin91, Bis93]. Selecting a set of values increases the chance that some of the inputs values will avoid the defect. For some of the simpler application functions, it is possible to define analytic transformations which use very different values from the intended input value (e.g. using the half angle formulae for sine and cosine function).

A similar concept was also proposed as a means of obtaining a reliable test 'oracle' [Lip91]. The method relies on identifying functions where it is possible to derive the same result using random offsets from the intended input value. For example, a linear function $P(x)$ could be calculated as $P(x + r) - P(r)$, where r is a randomly selected offset which can range over the whole input space. The function is then tested to establish a modest level of reliability using randomly selected numbers (e.g. 100 tests). Given some bound on the failure rate p for random execution, arbitrary levels of reliability can be obtained by repeated computation using randomized offsets followed by a majority vote. This reliability estimate is not affected by localized defects because the actual sample values are spread evenly over the entire input space, so n randomized computations which agreed imply that the result is correct with probability $1 - p^n$. The main problem is to identify functions that are *randomly testable* over the

whole input domain. This is not as restrictive as it first appears and it can be shown that conventional modulo 2 integer arithmetic (e.g. addition and multiplication) can be utilized, and also functions constructed using these basic operators (e.g. polynomials and the Fast Fourier Transform function). The method is also applicable to functions of rational numbers (and by implication floating point numbers) provided the precision is doubled to accommodate the inherent round-off effects. While this procedure has been proposed as a basis for generating off-line test data, it could also be used to implement on-line fault tolerance.

The data diversity strategy using very large random perturbations seems to be limited to 'well-behaved' programs or sub-programs where there is some concept of a continuous function over the whole input space. This could exclude important classes of programs such those which use Boolean logic, or switch between alternative functions under differing circumstances.

The use of small random perturbations about the input value combined with a voting mechanism could be effective, although the need for consistent comparison applies equally to data diversity as N-version diversity — so it would be most easily used on systems where the comparison requirements are minimal. The actual reliability enhancement achieved experimentally by such an approach depends on the shape and distribution of defects within the program. Recent work on defect shapes [Bis93] has indicated that, in a multi-dimensional space, the defect can have quite large internal dimensions even when the failure probability is small. However, empirical observations and a related theoretical argument suggest that the shapes will tend be defined by the program function (e.g. bounded by equivalence domain boundaries) and also suggest that the defect will be a hypersurface which is 'thin' in the direction normal to the surface. With sufficiently large perturbations it should in principle be possible to 'straddle' such defects.

Data diversity is an attractive concept since only a single version of the program is required. In its strongest form (using large perturbations) mathematically well-founded arguments can be made for very large reliability improvements (assuming the only faults are design faults). The main problem lies in extending its applicability over a sufficient wide range to be practically useful. The weaker form, using small perturbations, may only exhibit modest reliability improvements, but could be used in a range of practical applications. The author is not aware of any industrial application that uses this concept.

9.6 CONCLUSIONS

Almost from the beginning, it was recognized 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 (although there are some doubts about certain aspects of their experimental design).

This result is backed up by later experiments and qualitative evidence from a number of sources which show 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 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 favor 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 favor of N-version programming is that it can also give protection against dissimilar faults, so absolute fault-freeness is not required in any one version — 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, I would regard N-version programming as a useful safeguard against residual design faults, but my main concern is whether software diversity is justifiable in the overall system context. I would argue 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 utilize diversity at a higher level (e.g. a functionally diverse protective system) so that there is some defense-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.

REFERENCES

[Amm88] P.E. Amman and J.C. Knight. Data diversity: an approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4) April 1988.

[And81] H. Anderson and G. Hagelin. Computer controlled interlocking system. *Ericsson Review*, (2), 1981.

[Avi77] A. Avižienis and L. Chen. On the implementation of N-version programming for software fault tolerance during Execution. In *Proc. the First IEEE-CS International Computer Software and Applications Conference (COMPSAC 77)*, Chicago, November 1977.

[Avi88] A. Avižienis, M.R. Lyu and W. Schuetz. 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.

[Bet92] A.E. Betts and D. Wellbourne. Software safety assessment and the Sizewell B applications. *International Conference on Electrical and Control Aspects of the Sizewell B PWR*, 14-15 September 1992, Churchill College, Cambridge, organized by Institution of Electrical Engineers, IEE, 1992.

[Bis86] P.G. 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), 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. In *Proc. Safety and Reliability Society Symposium (SARSS 87)*, Manchester, Elsevier Applied Science, November 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. *Dependable Computing and Fault Tolerant Systems*, volume 4 of *Dependable Computing for Critical Applications*, (ed. A. Avižienis and J.C. Laprie), Springer Verlag, Wien-New York, 1991.

[Bis93] P.G. Bishop. The variation of software survival times for different operational profiles. In *Proc. FTCS 23*, Toulouse, June 1993. IEEE Computer Society Press.

[Blo86] R.E. Bloomfield and P.K.D. Froome. The application of formal methods to the assessment of high integrity software. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986.

[Bri89] S.S. Brilliant, J.C. Knight and N.G. Leveson. The consistent comparison problem. *IEEE Transactions on Software Engineering*, 15, November 1989.

[Bri90] S.S. Brilliant, J.C. Knight and N.G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[Bri93] D. Brire and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls — A family of fault-tolerant systems. In *Proc. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 616–623, Toulouse, France, June 1993. IEEE Computer Society Press.

[Con88] A.E. Condor and G.J. Hinton. Fault tolerant and fail-safe design of CANDU computerized 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. In *Proc. SAFECOMP 79*.

[Dun86] J.R. Dunham. Experiments in software reliability: life critical applications. *IEEE Transactions on Software Engineering*, SE-12(1), 1986.

[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 Transactions on Software Engineering*, SE-17(7):692–702, 1991.

[Eck85] D.E. Eckhardt and L.D. Lee. A theoretical basis for the analysis of multi-version software subject to coincident failures. *IEEE Transactions on Software Engineering*, SE-11(12), 1985.

[Fin91] G. Finelli. NASA software failure characterization experiments. *Reliability Engineering and System Safety*, 32, 1991.

[Gme80] L. Gmeiner and U. Voges. Software diversity in reactor protection systems: an experiment. In R. Lauber, editor, *Safety of Computer Control Systems*. Pergamon, New York, 1980.

[Gme88] G. Gmeiner and U. Voges. Use of diversity in experimental reactor safety systems. In U. Voges, editor, *Software Diversity in Computerized Control Systems*. Springer Verlag, 1988.

[Hag87] G. Hagelin. Ericsson system for safety control. In U. Voges, editor, *Software Diversity in Computerized Control Systems*. Springer Verlag, 1988.

[Kel83] J.P.J. Kelly and A. Avižienis. 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, and 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. *IEEE Transactions on Software Engineering*, SE-12, January 1986.

[Kni86b] J.C. Knight and N.G. Leveson. An empirical study of the failure probabilities in multi-version software. In *Proc. FTCS 16*, Vienna, July 1986.

[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 Transactions on Software Engineering*, SE-16(4), April 1990.

[Lev88] N.G. Leveson and T.J. Shimeall. An empirical exploration of five software fault detection methods. In *Proc. SAFECOMP 88*, Fulda Germany, November 1988.

[Lip91] R.J. Lipton. New directions in testing. *DIMACS Series in Discrete Mathematics and Computer Science* (American Mathematical Society), volume 2, 1991.

[Lit89] B. Littlewood and D. Miller. Conceptual modelling of coincident failures in multi-version software. *IEEE Trans on Software Engineering*, SE-15(12):1596–1614, 1989.

[Lyu92] M.R. Lyu. Software reliability measurements in an N-version software execution environment. *Proc. International Symposium on Software Reliability Engineering*, pages 254–263, October 1992.

[Lyu93] M.R. Lyu and Y. He. Improving the N-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189, June 1993.

[Lyu94] M.R. Lyu, J-H. Chen, A. Avižienis. Experience in metrics and measurements for N-version programming, *International Journal of Reliability, Quality and Safety Engineering*, 1(1), March 1994.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1, June 1975.

[Tra88] P. Traverse. Airbus and ATR system architecture and specification. In U. Voges, editor, *Software Diversity in Computerized Control Systems*. Springer Verlag, 1988.

[Vog82] U. Voges, F. French and L. Gmeiner. Use of microprocessors in a safety-oriented reactor shutdown system. In *Proc. EUROCON*, Lyngby, Denmark, June 1982.

[Vog94] U. Voges. Software diversity, *Reliability Engineering and System Safety*, (43), 1994.

[War93] N.J. Ward. Rigorous retrospective static analysis of the Sizewell B primary protection system software, *Proc. SAFECOMP 93*, Springer Verlag, 1993.