An Empirical Exploration of the Difficulty Function

Julian G W Bentley, Peter G Bishop, Meine van der Meulen

Centre for Software Reliability
City University
Northampton Square
London EC1V 0HB, UK

Abstract. The theory developed by Eckhardt and Lee (and later extended by Littlewood and Miller) utilises the concept of a "difficulty function" to estimate the expected gain in reliability of fault tolerant architectures based on diverse programs. The "difficulty function" is the likelihood that a randomly chosen program will fail for any given input value. To date this has been an abstract concept that explains why dependent failures are likely to occur. This paper presents an empirical measurement of the difficulty function based on an analysis of over six thousand program versions implemented to a common specification. The study derived a "score function" for each version. It was found that several different program versions produced identical score functions, which when analysed, were usually found to be due to common programming faults. The score functions of the individual versions were combined to derive an approximation of the difficulty function. For this particular (relatively simple) problem specification, it was shown that the difficulty function derived from the program versions was fairly flat, and the reliability gain from using multi-version programs would be close to that expected from the independence assumption.

1. Introduction

The concept of using diversely developed programs (N-version programming) to improve reliability was first proposed by Avizienis [1]. However, experimental studies of N-version programming showed that the failures of the diverse versions were not independent, for example [2, 4] showed that common specification faults existed, and Knight and Leveson [6] demonstrated that failure dependency existed between diverse implementation faults to a high level of statistical confidence. More generally, theoretical models of diversity show that dependent failures are likely to exist for any pair of programs. The most notable models have been developed by Eckhardt and Lee [5] and Littlewood and Miller [7]. A recent exposition of these theories can be found in [8]. These models predict that, if the "difficulty" of correct execution varies with the input value, program versions developed "independently" will, on average, not fail independently. A key parameter in these models is the "difficulty function". This function represents the likelihood that a randomly chosen program will fail for any given input scenario (i.e. the probability that the programmer is more likely to make a mistake handling this particular input scenario).

While there has been considerable theoretical analysis of diversity, and empirical measurement of reliability improvement, there has been little research on the direct measurement of the difficulty function. This paper presents an empirical analysis of many thousands of "independently" developed program versions written to a common specification in a programming contest. The objectives of the study were:

- to directly measure the failure regions for each program version,
- to examine the underlying causes for faults that lead to similar or identical failure regions,
- to compute the difficulty function by combining the failure region results
- to assess the average reliability improvement of diverse program pairs, and compare it with the improve expected if the failures were independent.

The focus of this study was on diverse *implementation* faults. The correctness, completeness and accuracy of the specification were considered to be outside the scope of this project. However, specification-related problems were encountered in the study, and are discussed later in the paper.

In Section 2 of the paper we describe the source of the program versions used in this study, Section 3 summarises the difficulty function theory, Section 4 describes the measurements performed on the programs, while Sections 5 and 6 present an analysis of the results. Sections 7 and 8 discuss the results and draw some preliminary conclusions.

2. The Programming Contest software resource

In the past, obtaining many independently developed program versions by different authors to solve a particular problem would have been difficult. However, with wider use of the Internet, the concept of "programming contests" has evolved. "Contest Hosts" specify mathematical or logical challenges (specifications) to be solved programmatically by anyone willing and able to participate. Participants make submissions of program versions that attempt to satisfy the published specification. These are then "judged" (usually by some automated test system at the contest site) and then accepted or rejected.

We established contact with the organiser of one of these sites (the University of Valladolid) which hosts contest problems for the ACM and additional contest problems maintained by the University [9]. The organiser supplied over six thousand program submissions for one of its published problems. The programs varied by author, country of origin, and programming language. Authors often submitted several versions in attempting to produce a correct solution to the problem. This program corpus formed the basis for our research study.

Clearly, there are issues about realism of these programs when compared to "real world" software development practices, and these issues are discussed in Section 7. However the availability of so many program versions does allow genuine statistical studies to be made, and does allow conjectures to be made which can be tested on other examples. In addition such conjectures can be evaluated on actual industrial software and hence have the potential to be extended to a wider class of programs.

3. Probability of failure and the difficulty function

Two of the most well known probability models in this domain, are the Eckhardt and Lee model [5], and, the Littlewood and Miller extended model [7]. Both models assume that:

1. Failures of an individual program π are deterministic and a program version either fails or succeeds for each input value x. The failure region of a program π can be represented by a "score" function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails.
2. There is randomness due to the development process. This is represented as the random selection of a program from the set of all possible program versions Π that can feasibly be developed and/or envisaged. The probability that a particular version π, will be produced is $P(\pi)$.
3. There is randomness due to the demands in operation. This is represented by the (random) set of all possible demands X (i.e. inputs and/or states) that can possibly occur, together with the probability of selection of a given input demand x, $P(x)$.

Using these model assumptions, the average probability of a program version failing on a given demand is given by the *difficulty function*, $\theta(x)$, where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) P(\pi) \tag{1}$$

The average probability of failure per demand (*pfd*) of a randomly chosen single program version can be computed using the difficulty function and the demand profile $P(x)$:

$$E(\mathrm{pfd}_1) = \sum_{x} \theta(x) P(x) \tag{2}$$

The average pfd of randomly chosen pair of program versions (π_A, π_B) taken from two possible populations A and B is:

$$E(\mathrm{pfd}_2) = \sum_{x} \theta_A(x) \theta_B(x) P(x) \tag{3}$$

The Eckhardt and Lee model assumes similar development processes for A and B and hence identical difficulty functions

$$E(\mathrm{pfd}_2) = \sum_{x} \theta(x)^2 P(x) \tag{4}$$

where $\theta(x)$ is the common difficulty function. If $\theta(x)$ is constant for all x (i.e. the difficulty function is "flat") then, the reliability improvement for a diverse pair will (on average) satisfy the independence assumption, i.e.:

$$E(\mathrm{pfd}_2) = E(\mathrm{pfd}_1)^2 \tag{5}$$

However if the difficulty function is "bumpy", it is always the case that:

$$E(\mathrm{pfd}_2) \geq E(\mathrm{pfd}_1)^2 \tag{6}$$

If there is a very "spiky" difficulty surface, the diverse program versions tend to fail on exactly the same inputs. Consequently, diversity is likely to yield little benefit and pfd_2 is close to pfd_1. If, however, there is a relatively "flat" difficulty surface the program versions do not tend to fail on the same inputs and hence pfd_2 is closer to pfd_1^2 (the independence assumption).

If the populations A and B differ (the Littlewood and Miller model), the improvement can, in principle, be *better* that the independence assumption, i.e. when the "valleys" in $\theta_A(x)$ coincide with the "hills" in $\theta_B(x)$, it is possible for the expected pfd_2 to be less than that predicted by the independence assumption.

4. Experimental study

For our study we selected a relatively simple Contest Host problem. The problem specified that two inputs, velocity (v) and time (t) had to be used to compute a displacement or distance (d). The problem had defined integer input ranges. Velocity v had a defined range of ($-100 \leq v \leq 100$), whilst time t was defined as ($0 \leq t \leq 200$). A set of 40401 unique values would therefore cover all possible input combinations that could be submitted for the calculation. However, this was not the entire input domain, because the problem specification permitted an arbitrary sequence of input lines, each specifying a new calculation. If all possible sequences of the input pairs (v, t) were considered, assuming no constraints on sequencing or repetition, the input domain for the program could be viewed as infinite. However, as each line of input should be computed independently from every other line, the sequence order should not be relevant, so the experiment chose to base its analysis on the combination of all possible values of v and t. This can be viewed as a projection of the input domain (which has a third "sequence" dimension) on to the (v, t) plane.

The experiment set up a test harness to apply a sequence of 40401 different values of v and t to the available versions. The results for each version were recorded and compared against a selected "oracle" program. The success or failure of each input could then be determined. Some versions were found to have identical results to others for all inputs. The identical results were grouped together in "equivalence classes".

In terms of the difficulty function theory outlined, each equivalence class was viewed as a possible program, π, taken from the universe of all programs, Π, for that specification. The record of success/failure for each input value is equivalent to the score function, $\omega(\pi, x)$ for the equivalence class as it represents a binary value for every point in the input domain, x, indicating whether the result was correct or not. For the chosen problem, the input domain, x, is a two-dimensional space with axes of velocity (v) and time (t), and the score function represented the failure region within that input domain.

$P(\pi)$ was estimated by taking the ratio of the number of instances in an equivalence class against the total number of programs in the population. The size of the failure region was taken to be the proportion of input values that resulted in failure. The failure regions can be represented two dimensionally on the v, t plane, but it should be emphasised that this is only a projection of the overall input domain. It is only possible to sample the total input domain.

5. Results

The results revealed that the 2529 initial program versions produced by the authors (the "v1" population) formed 50 equivalence classes. The five most frequent equivalence classes accounted for approximately 96% of the population. The results of the analysis are summarised in Table 1.

Table 1. Population v1 equivalence classes (frequent)

Equivalence Class (π)	Number of versions	P(π)	Size of Failure Region
EC1	1928	0.762	0.000
EC2	201	0.079	1.000
EC3	189	0.075	0.495
EC4	90	0.036	0.999
EC5	27	0.011	0.990

Equivalence class 1 agrees with the oracle program. There are no known faults associated with this equivalence class result, consequently the size of the failure region was 0%.

For equivalence class 2, analysis of the programs revealed a range of different faults resulted in complete failure across the input domain.

For equivalence class 3, failures always occurred for $v < 0$. This was due to a specification discrepancy on the Contest Host web site. Two specifications existed on the site—one in a PDF document, the other on the actual web page. The PDF specification required a *distance* (which is always positive) while the web specification required a *displacement* which can be positive or negative. The "displacement" version was judged to be the correct version.

Equivalence class 4, typified those versions that lacked implementation of a loop to process a sequence of input lines (i.e. only computed the first input line correctly).

For equivalence class 5, inspection of the program versions revealed a variable declaration fault to be the likely cause.

A similar analysis was performed on the final program version submitted by each author (the "vFinal" population). The results revealed that of the 2666 final program versions could be grouped into 34 equivalence classes. The five most frequent equivalence classes accounted for approximately 98% of the population. The results of the analysis are summarised in Table 2.

Table 2. Population vFinal: equivalence classes

Equivalence Class (π)	Number of versions	P(π)	Size of Failure Region
EC1	2458	0.922	0.000
EC2	70	0.026	1.000
EC3	40	0.015	0.495
EC4	21	0.008	0.999
EC5	13	0.005	0.990

Note that there is some overlap between the "first" and "final" populations as some authors only submitted one version. It can be seen that the dominant equivalence classes are the same as in the first version, but the proportions of each equivalence class have decreased (apart from EC1) presumably because some programs have been successfully debugged.

Figure 1 shows examples of the less frequent equivalence class failure regions.

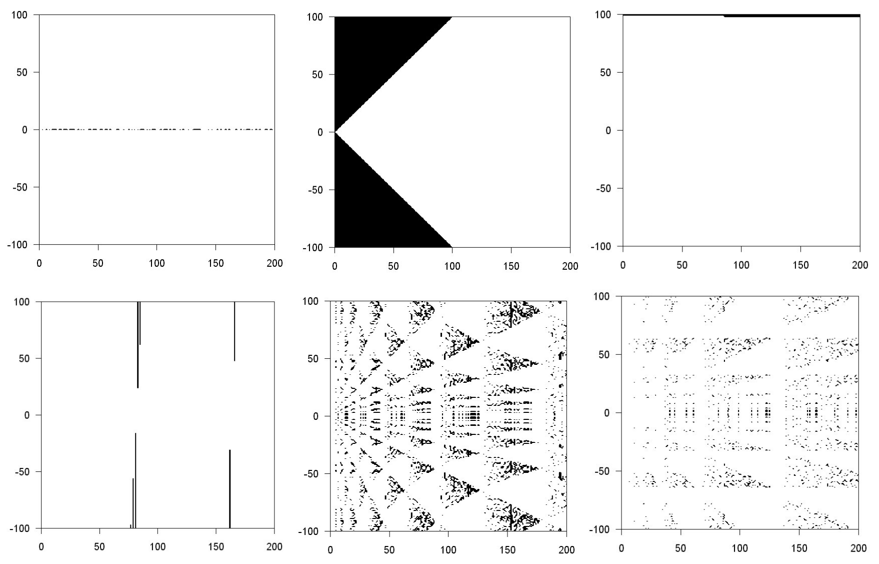

Fig. 1. Failure regions for some of the infrequent equivalence classes

These graphs show that there is a remarkable variation in the failure regions even for such a simple problem. The failure regions for the frequent EC1 to EC5 equivalence classes are simpler in structure, i.e. all "white" for EC1, almost all "black" for EC2, EC4 and EC5 and a black rectangle for EC3 covering the negative portion of the input domain.

6. Analysis

The “score functions” and frequency data of the equivalence classes can be combined to estimate the difficulty function for the specific problem. Note that this is an approximation to the actual difficulty function which should be an average taken over the population of all possible programs. It is unlikely that the set of all possible programs (Π) are limited to the 50 equivalence classes identified in this study. However, a computation of $\theta(x)$ based on the known equivalence classes should give a good approximation to the difficulty function, as 95% (v1) and 97% (vFinal) of the program versions belonged to four of the most frequently occurring known equivalence classes so uncertainties in the “tail” of the population of programs will only have a marginal effect on the difficulty function estimate.

One issue that needed to be considered in the analysis was the effect of the specification discrepancy. The discrepancy will bias the estimate of implementation difficulty as equivalence class EC3 might not have occurred if the specification had been unambiguous. On the other hand, such specification problems might be typical of the effect of specification ambiguity on the difficulty function. We therefore calculated the difficulty function in two ways:

- including all equivalence classes
- all equivalence classes except EC3 (the adjusted difficulty function).

6.1 Calculation of the difficulty function

For each input value, x, the difficulty function value $\theta(x)$ was estimated using equation (1) and the result for the v1 population is shown in Figure 2.

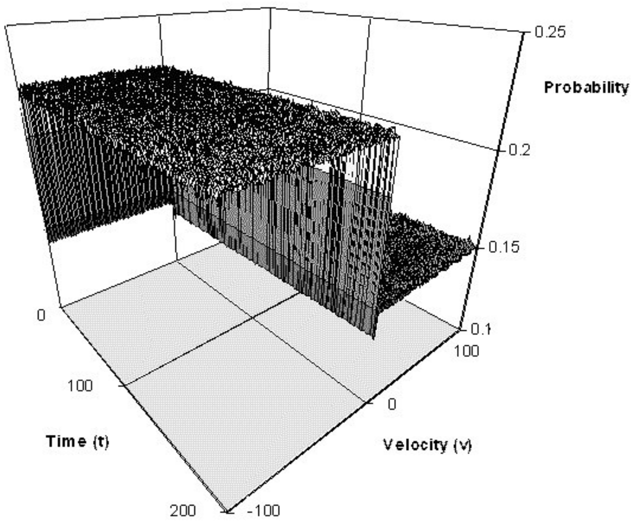

Fig. 2. Difficulty function for the v1 population

This calculation assumes that π is the same as an equivalence class, the score function $\omega(\pi, x)$ is the same as the observed failure region and $P(\pi)$ is the relative frequency of the equivalence class in the population. Effectively the calculation takes a weighted average of the individual failure regions in the v1 population.

Note that the difficulty function shown in Figure 2 has not accounted for any bias introduced by the specification discrepancy and the "step" in difficulty for $v < 0$ is due to the specification ambiguity.

The probability of failure also decreases for certain "special" values—the velocity axis v=0, and time axis t=0. This might be expected since an incorrect function of v and t might well yield the same value as the correct function for these special values (i.e. a displacement of zero). There is also a low probability value at v=-100, t=0 which is due to faults that fail to execute subsequent lines in the input file, and the first test input value happens to be v=-100, t=0. If the test input values had been submitted in a random order, this point would have been no more likely to fail than adjacent points.

It can also be seen that there is a certain amount of "noise" on the two "flat" regions of the difficulty surface. This is caused by some of the highly complex failure patterns that exist for some of the infrequent equivalence classes (as illustrated in Figure 1).

The results were adjusted to account for specification bias by eliminating the equivalence class EC3 and Figure 3 shows the adjusted difficulty function.

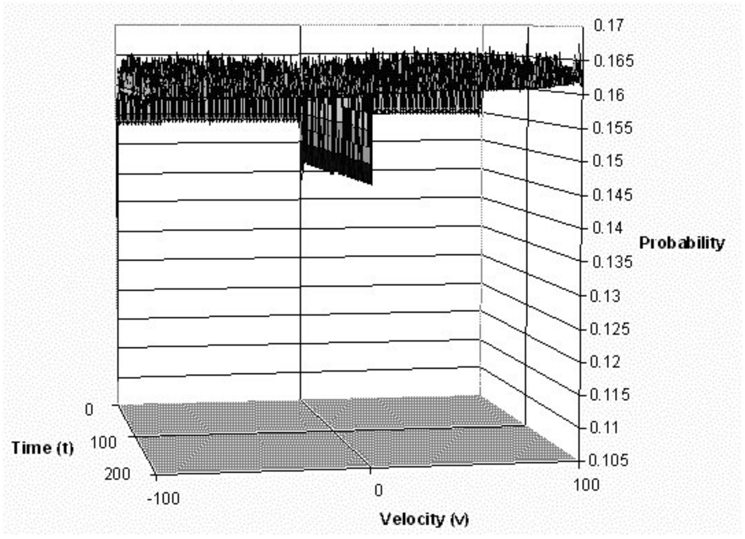

Fig. 3. Adjusted difficulty function for the v1 population

With the adjustment for specification bias, the difficulty function is now almost "flat" apart from the "special case" values on the velocity axis, v=0, and time axis, t=0.

The difficulty functions for the final version populations (adjusted and unadjusted) are very similar in shapes observed in the v1 population. The adjusted vFinal difficulty function is shown in Figure 4.

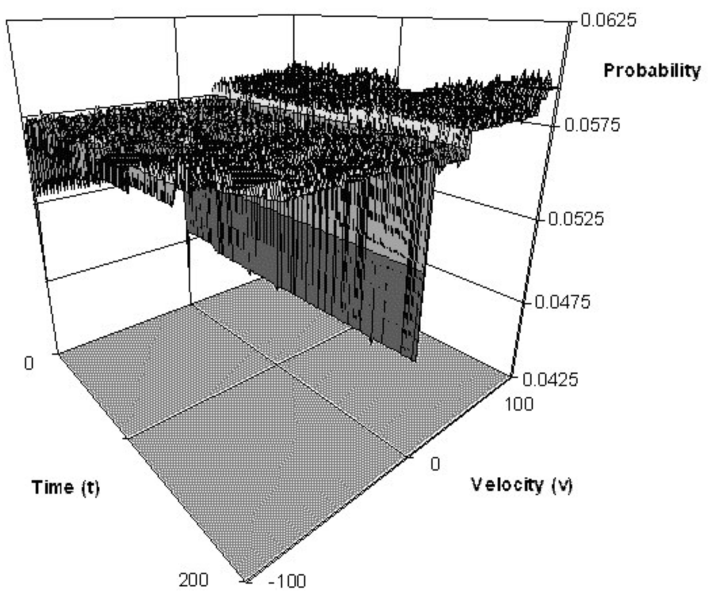

Fig. 4. Adjusted difficulty function for the vFinal population

While the difficulty functions are similar in shape to the v1 population difficulty functions, the mean value of $\theta(x)$ is about one third that of the v1 population—mainly because the vFinal population contains a higher proportion of correct program versions.

The mean values for $\theta(x)$ are summarised in the table below.

Table 3. Mean difficulty values for different program populations

Program Population	Mean Value of θ	
	Unadjusted	Adjusted
v1	0.186	0.161
vFinal	0.064	0.058

6.2 Expected *pfd* of a single version and a pair of versions

To compute the *pfd* for an average program from equation (2), we need to know the execution profile P(x). This could vary from one application context to another. However, assuming any input is equally likely, the *pfd* of a single version is the mean value of θ, while the dangerous failure rate of a fault-detecting pair, pfd_2, given in equation (4) reduces to the mean of $\theta(x)^2$ averaged over the input space. Note that this assumes the same difficulty function for both programs, i.e. they are drawn from the same population (the Eckhardt and Lee assumption [5]).

The expected *pfd*s for a single version and a pair of versions were computed for the v1 and vFinal program populations (and the adjusted versions). The results are shown in the table below and compared with the pfd expected from the independence assumption (pfd_1^2).

Table 4. Comparison of expected probability of failure on demand

Program Population	pfd_1	pfd_2	pfd_1^2
v1	0.186	0.0361	0.0347
v1 adjusted	0.161	0.0260	0.0260
vFinal	0.064	0.0042	0.0041
vFinal adjusted	0.058	0.0033	0.0033

The increase in the *pfd* of a diverse pair (pfd_2) relative to the independence assumption (pfd_1^2) was relatively small for all populations, and for the adjusted populations, the difference between pfd_2 and the independence assumption is almost negligible. The worst-case increase relative to the independence assumption was observed to be 1.04 (for the unadjusted v1 population). This is consistent with expectations, as the difficulty surface was much flatter for the adjusted versions.

7. Discussion

While the results are interesting, we have to be cautious about their applicability to "real world" programs. Programming contests can provide many thousands of versions and this is a clear benefit for statistical studies. On the other hand, the results may be unrepresentative of software development in industry, especially in that:

1. many of the developers are probably amateurs or students rather than professional developers;
2. the program specifications are not overly complex, so that the programs are not typical of software developed in industry, and whole classes of faults that arise in the development of complex software may be missing;
3. the development process is different from the processes applied in industry;
4. there is no experimental control over program development, so independence could be compromised, e.g. by copying other participants' programs, or by submitting programs produced collectively by multiple people.

Discussions with the contest organiser suggests that plagiarism is not considered to be a major issue and, in any case, the main effect of plagiarism of correct versions would be to increase the number of correct versions slightly. In principle, it should be feasible to trap programs from different authors that are identical or very similar in structure.

With regard to programming expertise, the top participants are known to take part in international programming contests under controlled conditions (in a physical location rather than on the internet). So it seems that there is a very broad range of expertise. In future studies we might be able to obtain more information about the participants so that the level of expertise can be more closely controlled.

The example we have studied is "programming in the small" rather than "programming in the large". It is therefore likely that there are classes of "large program" fault, such as integration and interface faults, that will not be present in our example. In addition the processes used differ from industry practice. However experience with quite large programs indicates that many of the faults are due to localised programming errors that remained undetected by industrial verification and validation phases. So it is likely the errors committed in small contest-derived programs will also arise in large industrial programs, although we have to recognise that the set of faults will be incomplete and the relative frequency of the fault classes is likely to differ.

From the foregoing discussion, it is clear that we cannot make general conclusions from a single program example. However, the results can suggest hypotheses to be tested in subsequent experiments. One clear result of this experiment is that the difficulty surface is quite flat. The specification required a single simple "transfer function" that applies to the whole of the input domain. One might conjecture that, for a fixed transfer function, the difficulty would be the same for all input values. Similarly where the program input domain is divided into sub-domains which have different transfer functions, we might expect the difficulty to be flat within each sib domain. If this conjecture is correct, we would expect diverse programs with simple transfer functions to have reliability improvements close to that predicted by independent failure assumption. Indeed, diversity may be better suited to simple functions rather than entire large complex programs. However we emphasise that this is a conjecture, and more experiments would be needed to test this hypothesis.

It should also be noted the *pfd* reduction derived in Table 4 is the *average reduction*. For a specific pair of program versions it is possible for the actual level of reduction to vary from zero to complete reduction. A zero reduction case would occur if a pair of versions from the same equivalence class are selected. Conversely, complete reduction occurs if an incorrect version is combined with a correct version. In Table 2, for instance, 92% of versions in the final population are correct so the chance that a pair of versions will be faulty is $(1-0.92)^2$, i.e. 0.64%. It follows that the chance of a totally fault detecting pair (where at least one version is correct) will 99.36%. Pairs with lower detection performance will be distributed within in the remaining 0.64% of the population of possible pairs and some of these versions will behave identically and hence have zero failure detection probability.

Another issue that we did not plan to examine was the impact of specification problems. However it is apparent that the problem we encountered was a particular example of specification ambiguity that arises in many projects. This illustrates how N-version programming can be vulnerable to common specification problems, and the need for appropriate software engineering strategies to ensure that specifications are sound.

At a more general level, we have to ask whether such experiments are of practical relevance to industry. As discussed earlier, the examples we use are not typical as they are not as complex as industrial software and the development processes differ. However, the experiments could lead to conjectures that *could* be tested on industrially produced software, (such as the assumption of constant difficulty over a sub-domain). If such conjectures are shown to be applicable to industrial software, this information could be used to predict, for example, the expected *variation* in difficulty over the sub-domains and hence the expected gain from using diverse software. It has to be recognised that relating the research to industrial software will be difficult and, at least initially, is most likely to be applicable to software implementing relatively simple functions (like smart sensors). We hope to address this issue in future research.

8. Conclusions and further work

We conclude that:

1. One significant source of failure was the specification. We were able to allow for the specification discrepancy in our analysis, but it does point to a more general issue with N-version programming, i.e. that it is vulnerable errors in the specification, so a sound specification is an essential prerequisite to the deployment of N-version programming.
2. For this particular example, the difficulty surface was almost flat. This indicates that there was little variation of difficulty and a significant improvement in reliability should (on average) be achieved, although the reliability of arbitrary pair of versions can vary significantly from this average.

We conjecture that for programs with a single simple transfer function over the whole input domain (like this example), the difficulty function might turn out to be relatively flat. In that case, reliability improvements close to the assumption of independent failures may be achievable. However, more experiments would be needed to test this hypothesis.

There is significant potential for future research on variations in difficulty. The possibilities include:

1. Variation of difficulty for *different* sub-populations (e.g. computer language, author nationality, level of expertise, etc). The extended Littlewood and Miller theory suggests it is possible to have reliability better than the independence assumption value. An empirical study could be envisaged, to determine if this is observed when versions from different populations are combined.
2. Extension to other contest host program examples, and more wide-ranging experiments to assess conjectures like the flat difficulty conjecture discussed above.
3. Relating the hypotheses generated in the experiments to industrial examples.

Acknowledgements

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the *Interdisciplinary Collaboration on the Dependability of computer based systems*, (DIRC), and via the *Diversity with Off-The-Shelf Components* Project (DOTS), GR/N24056. The authors would like to thank Miguel Revilla, University of Valladolid, Spain, for providing the program versions and information relating to the programming contest.

References

[1] 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.

[2] A. Avizienis and J. P. J. Kelly, “Fault Tolerance by Design Diversity: Concepts and Experiments”, Computer, Vol. 17, No. 8, August 1984.

[3] A. Avizienis, “Software Fault Tolerance”, Information Processing (G.X. Titter, Ed.), pp. 491-498, Elsevier Science Publishers, Holland, 1989.

[4] P.G. Bishop, M. Barnes, et/ al., “PODS a Project on Diverse Software”, IEEE Trans. Software Engineering, Vol. SE-12, No. 9, 929-940, 1986.

[5] D. E. Eckhardt, L. D. Lee, “A theoretical basis for the analysis of multiversion software subject to coincident errors”, IEEE Transactions on Software Engineering, SE-11 (12), pp.1511-1517, 1985.

[6] J. C. Knight and N. G. Leveson, “An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming”, IEEE Transactions on Software Engineering, SE-12 (1), pp. 96-109, 1986.

[7] B. Littlewood and D. R. Miller, “Conceptual Modelling of Coincident Failures in Multiversion Software”, IEEE Transactions on Software Engineering, Vol. 15, No. 2, pp. 1596-1614, December 1989.

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

[9] S. Skiena and M. Revilla, Programming Challenges, ISBN: 0387001638, Springer Verlag, March, 2003, (http://acm.uva.es/problemset/)