The Effectiveness of Choice of Programming Language as a Diversity Seeking Decision

Meine J.P. van der Meulen[1] and Miguel Revilla[2]

[1] Centre for Software Reliability, City University, London
`http://www.csr.city.ac.uk`
[2] Department of Applied Mathematics,
University of Valladolid, Spain
`http://www.mac.cie.uva.es/~revilla/`

Abstract. Software reliability can be increased by using a diverse pair of programs (1-out-of-2 system), both written to the same specification. The improvement of the reliability of the pair versus the reliability of a single version depends on the degree of diversity of the programs. The choice of programming language has been suggested as an example of a diversity seeking decision. However, little is known about the validity of this recommendation. This paper assesses the effect of language on program diversity.

We compare the effects of the choice of programming language as a diversity seeking decision by using programs written to three different specifications in the "UVa Online Judge". Thousands of programs have been written to these specifications; this makes it possible to provide statistical evidence.

The experiment shows that when the average probability of failure on demand (pfd) of the programs is high, the programs fail almost independently, and the choice of programming language does not make any difference. When the average pfd of the pools gets lower, the programs start to fail dependently, and the pfd of the pairs deviates more and more from the product of the pfds of the individual programs. Also, we observe that the diverse C/Pascal or C++/Pascal pairs perform as good as or better than the other possible pairs.

1 Introduction

The use of a diverse pair of programs has often been recommended to achieve high reliability [1] [2] [3] [4] [5]. Software diversity may however not lead to a dramatically high improvement. This is caused by the fact that the behaviour of the programs cannot be assumed to be independent [3] [5] [6] [7]. Two program versions written by independent teams can still contain similar programming mistakes, thus limiting the gain in reliability of the diverse pair.

In spite of this, the case for diversity for achieving high reliability remains strong. The possible gain using diversity appears to be higher than can be achieved by trying to write a high reliability single program [6].

Several techniques have been proposed to decrease the likelihood that different programs fail dependently. These are called "Diversity Seeking Decisions" in [9]. Examples are:

- **Data diversity.** Using random perturbations of inputs; using algorithm specific re-expression of inputs.
- **Design diversity.** Separate ("independent") development; diversity in programming language; diverse requirements/specifications; different expressions of identical requirements; etc.

In this paper we will concentrate on design diversity and specifically on programming language diversity. This is a potential defence against some programming slips, and provides some, limited, cognitive diversity against mistakes in higher-level problem solving; the efficacy will however depend heavily on "how different" the programming languages are.

The "UVa Online Judge"-website (`http://acm.uva.es`) provides many programs written to many specifications, and gives us the opportunity to compare diverse pairs. In this research we use the programs written in C, C++ and Pascal, written to three different specifications. Our aim is to compare the reliability performance of diverse pairs with each other and with single programs.

2 The Experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website is an initiative of Miguel Revilla of the University of Valladolid [10]. It contains problems to which everyone can submit solutions. The solutions are programs written in C, C++, Java or Pascal. The correctness of the programs is automatically judged by the "Online Judge". Most authors submit solutions until their solution is judged as being correct. There are many thousands of authors and together they have produced more than 3,000,000 solutions to the approximately 1500 problems on the website.

In this paper we will analyse the programs written to three different problems on the website. We will submit every program to a test set, and then compare their failure behaviour.

There are some obvious drawbacks from using this data as a source for scientific analysis. First of all, these are not "real" programs: the programs under consideration solve small, mostly mathematical, problems. We have to be careful to not overinterpret the results.

Another point of criticism might be the fact that the Online Judge does not give feedback on the demand on which the program failed. This is not necessarily a drawback. It is certainly not comparable to a development process involving a programmer and a tester, because in that case there will be feedback on the input data for which the program fails. It has however similarities with a programmer's normal development process: a programmer will in spite of the fact that there are no examples of inputs for which a program fails, assume that it is not yet correct. The programmer works until he is convinced that the program is correct, based on his own analysis and testing. From this perspective, the Online Judge only confirms the programmer's intuition that the program is not yet correct. In this experiment, we circumvent this drawback by only using first submissions.

A last possible criticism on our approach is that programmers may copy each other's results. This may be true, but it is possible to limit the consequences of this plagiarism for the analyses by assuming that authors will only copy correct results from each other. For the analyses in this paper, the consequence is that we cannot trust absolute results, and we will limit ourselves to observing trends in relative performance.

2.2 Problems Selected

We selected problems from conforming to the following criteria:

- The problem does not have history, i.e. subsequent inputs should not influence each other. Of course, some programmers may implement the problem in such a way that it has history. Given our test approach, see below, we will not detect these kinds of faults.
- The problem has a limited demand space: two integer inputs.

Both restrictions lead to a reduction of the size of the demand space and this keeps the computing time within reasonable bounds (the necessary preparatory calculations for the analysis of these problems take between a day and two weeks to complete).

Below, we provide a short description of the problems, although this information is not necessary for reading this paper: we will not go into detail with respect to functionality. It gives some idea of the nature and difficulty of the problems, which is useful for interpreting our results. See the website`http://acm.uva.es` for more detailed descriptions of the problems.

The "3n+1"-Problem. A number sequence is built as follows: start with a given number; if it is odd, multiply by 3 and add 1; if it is even, divide by 2. The sequence length is the number of these steps to arrive at a result of 1. Determine the maximum sequence length for the numbers between two given integers $0 < i, j \leq 100,000$.

The "Factovisors"-Problem. For two given integers $0 \leq i, j \leq 2^{31}$, determine whether j divides factorial i.

The "Prime Time"-Problem. Euler discovered that the formula $n^2 + n + 41$ produces a prime for $0 \leq n \leq 40$; it does however not always produce a prime. Write a program that calculates the percentage of primes the formula generates for n between two integers i and j with $0 \leq i \leq j \leq 10,000$.

2.3 Running the Programs

For all problems chosen, a "demand" is a set of two integer inputs. Every program is restarted for every demand; this is to ensure the experiment is not influenced by history, e.g. when a program crashes for certain demands. We set a time limit on each demand of 200 ms. This time limit is chosen to terminate programs that are very slow, stall, or have other problems.

Table 1. Some statistics on the three problems

	3n+1			Factovisors			Prime Time		
	C	C++	Pas	C	C++	Pas	C	C++	Pas
Number of authors	5897	6097	1581	212	582	71	467	884	183
First attempt correct	2483	2442	593	113	308	42	356	653	127
First version completely incorrect	723	761	326	27	97	9	93	194	49

Every program is submitted to a series of demands. The outputs generated by the programs are compared to each other. Programs that produce exactly the same outputs form an "equivalence class". These equivalence classes are then converted into score functions. A score function indicates which demands will result in failure. The difference between an equivalence class and its score function is that programs that fail in different ways (i.e. different, incorrect outputs for the same demands) are part of different equivalence classes; their score functions may however be the same. The score functions are used for the calculations below.

For all three problems, we chose the equivalence class with the highest frequency of occurrence as the oracle, i.e. the version giving all the correct answers.

"3n+1" and "Factovisors" were run using the same set of demands: two numbers between 1 and 50, i.e. a total of 2500 demands. In both cases the outputs of the programs were deemed correct if they exactly match those of the oracle.

"Prime Time" was run using a first number between 0 and 79, and a second number between the first and 79, i.e. a total of 3240 demands. The outputs of the programs were deemed correct if they were within 0.01 of the output of the oracle, thus allowing for errors in round off (the answer is to be given in two decimal places).

Table 1 gives some statistics on the problems.

3 Effectiveness of Diversity

Two of the most well known probability models in this domain are the Eckhardt and Lee model [3] and the Littlewood and Miller extended model [7]. Both models assume that:

1. Failures of an individual program are deterministic and a program version either fails or succeeds for each input value x. The failure set 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, Π, from the set of all possible program versions that can feasibly be developed and/or envisaged. The probability that a particular version π will be produced is $P(\Pi = \pi)$. This can be related to the relative numbers of programs in the pools.
3. There is randomness due to the demands in operation. This is represented by the random occcurrence of a demand, X, from the set of all possible

demands. The probability that a particular demand will occur is $P(X = x)$, the demand profile. In this experiment we assume a contiguous demand space in which every demand has the same probability of occurring.

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

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

The average probability of failure on demand (pfd) of a randomly chosen single program version, can be computed using the difficulty function and the demand profile, in our case for two program versions, Π_A and Π_B:

$$\mathrm{pfd}_A := \sum_{x} \theta_A(x) P(X = x), \quad \mathrm{pfd}_B := \sum_{x} \theta_B(x) P(X = x) \tag{2}$$

The Eckhardt and Lee model assumes similar development processes for the two programs A and B and hence identical difficulty functions: $\theta_A(x) = \theta_B(x)$. So the average pfd for a pair of diverse programs (assuming the system only fails when both versions fail, i.e. a 1-out-of-2 system) would be:

$$\mathrm{pfd}_{AB} := \sum_{x} \theta_A(x)\theta_B(x) P(X = x) = \sum_{x} \theta_A(x)^2 P(X = x) \tag{3}$$

And:

$$\mathrm{pfd}_{AB} = \mathrm{pfd}_A^2 + var_X(\theta_A(X)) \tag{4}$$

The Littlewood and Miller model does not assume that the development processes are similar, and thus allows the difficulty functions for the two versions, $\theta_A(x)$ and $\theta_B(x)$, to be different. Therefore, the probability of failure of a pair remains:

$$\mathrm{pfd}_{AB} := \sum_{x} \theta_A(x)\theta_B(x) P(X = x) \tag{5}$$

And:

$$\mathrm{pfd}_{AB} = \mathrm{pfd}_A.\mathrm{pfd}_B + cov_X(\theta_A(X),\ \theta_B(X)) \tag{6}$$

In this experiment, we wish to investigate the reliability improvement gained by choosing different programming languages for the programs in the pair, and we therefore need to use the Littlewood and Miller model.

First, we establish pools of programs, each pool containing programs in C, C++ or Pascal. For comparison of the individual programs and pairs, we need pools with the same pfd. To manipulate the pfd of the pools, we remove programs from them, starting with those with the highest pfd, until the average pfd of the pool has the desired value. This is a possible way of simulating testing of the programs; the tests remove the programs with the highest pfd first. Pools with the same pfd could then be assumed to have undergone the same scrutiny of testing.

We select a first program from one of the pools. Then we select a second program from a pool, and calculate the ratio of the pfd of the first program and the pfd of the pair:

$$R = \frac{\mathrm{pfd}_A}{\mathrm{pfd}_{AB}} = \frac{\sum_x \theta_A(x)P(X=x)}{\sum_x \theta_A(x)\theta_B(x)P(X=x)} \tag{7}$$

We do so for varying values of the pfd of the pools. The varying pfd is shown on the horizontal axis in the graphs.

Figures 1, 2 and 3 show these ratios for the three problems for different choices of the programming language of the first program. The graphs show R on the vertical axes on a logarithmic scale, because we are interested in the reliability improvement; with a logarithmic scale, equal improvements have equal vertical distance.

4 Analysis and Discussion

All graphs clearly show that for pairs of programs with higher pfds (pfd$> 10^{-1.5}$) the choice of programming language does not make any difference. The pfd of the pair is fairly close to the product of the pfds of the individual programs. This indicates that the programs fail almost independently.

For the lower pfds (pfd$< 10^{-2}$) the pfd of the pair deviates more and more from the product of the pfds of the individual programs, the programs fail dependently. The reliability improvement reaches a "plateau" at between one to two orders of magnitude better than a single version. This is in accordance with the generally accepted assumption that the gain from redundancy is limited, and is certainly not simply the product of the pfds of the individual programs (e.g. in IEC61508 [13] the reliability improvement one can claim for applying redundancy is one SIL, i.e. a factor of 10; Eckhardt and Lee reach a comparable conclusion in [4]).

If we now compare the effect of choice of programming language in the pairs, we can observe that the C/Pascal and the C++/Pascal pairs almost always outperform the other pairs, most notably also the C/C++ pairs. The effect is most clearly visible in the "Factovisors"-problem, and in the middle region (10^{-3} < pdf < 10^{-2}) of the "3n+1"-problem. It is also visible in the "Prime Time"-problem, but in this case it is only visible in a small, unreliable, region of the graph. This graph is unreliable for pfd$< 10^{-2.5}$, because the pool of Pascal programs is almost empty.

The Littlewood and Miller model provides a description of diversity, in which the reliability change compared to the independence assumption is given by a covariance term, see Equation 6. Since the model cannot predict the shapes of the difficulty functions in C, C++ or Pascal, the model can not predict how large this change will be. The model however provides an intuition that when programmers make different faults in different programming languages their respective difficulty functions will be different. These differences could then lead to a better performance of diverse language pairs.

Analysis of the differences in programming faults of the "3n+1"-problem shows that faults in programming "for"-loops in Pascal are rare compared to C and C++. This accounts for the reliability improvement of Pascal/C and Pascal/C++ pairs in the middle pfd-region.

In the low-pfd region, pfd $< 10^{-4}$, we observe for "3n+1" that the reliability improvements of the pairs become approximately the same, whereas for "Factovisors" the diverse language pairs (C/Pascal and C++/Pascal) show an enormous improvement, even approaching infinity for the lowest pfd-values. This observation gives rise to some thoughts[1]

First, these observations in the low-pfd region confirm the theoretical result of the Littlewood and Miller model that the reliability of a diverse pair can be better than under the independence assumption.

Second, why do we observe this effect? As explained above, the pfd of a pool is established by subsequently removing the most unreliable programs. For low pfd-values this approach leads to a monoculture, and in the end only few different program behaviours will be present in the pool. In the case of "3n+1" these program behaviours happen to be roughly the same. In the case of "Factovisors" however, the last programs in the Pascal pool fail on different demands than those left in the C and C++ pools, thus leading to an enormous improvement in the reliability of the pair.

This result should be considered with care, because it could be an artefact of our experiment, caused by the way in which we establish a given pfd for a pool. On the other hand, a normal debugging process will have a similar effect: eliminating some behaviours, starting with the most unreliable ones, thus eventually also leading to a monoculture of behaviours.

5 Conclusion

We analysed the effect of the choice of programming language in diverse pairs using three different problems from the "UVa Online Judge". The results seem to indicate that diverse language pairs outperform other pairs, but the evidence is certainly not strong enough for a definite conclusion. Analysis of more problems could help to strengthen the evidence and also to identify the factors that influence the gain possibly achieved by diversity of programming language.

Acknowledgement

This work is supported by EPSRC/UK in the framework of the *Interdisciplinary Research Collaboration in Dependability* Project (DIRC).

[1] It has to be noted here that the amount of Pascal programs in this pfd-region for "Factovisors" is rather low, and the graph has to taken cum grano salis.

References

1. Brilliant, S.S., J.C. Knight, N.G. Leveson, *Analysis of Faults in an N-Version Software Experiment*, IEEE Transactions on Software Engineering, SE-16(2), pp. 238-47, February 1990.
2. Voges, U., *Software diversity*, Reliability Engineering and System Safety, Vol. 43(2), pp. 103-10, 1994.
3. Eckhardt, D.E., L.D. Lee, *A Theoretical Basis for the Analysis of Multi-Version Software Subject to Coincident Errors*, IEEE Transactions on Software Engineering, Vol. SE-11(12), pp. 1511-1517, December 1985.
4. Eckhardt, D.E., A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, J.P.J. Kelly, *An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*, IEEE Transaction on Software Engineering, Vol. 17, No. 7, July 1991.
5. Knight, J.C., N.G. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*, IEEE Transaction on Software Engineering, Vol. SE-12(1), pp. 96-109, 1986.
6. Hatton, L., *N-Version Design Versus One Good Version*, IEEE Software, 14, pp. 71-6, 1997.
7. Littlewood, B., 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. Lyu, M.R., *Software Reliability Eningeering*, McGraw Hill, 1995.
9. Popov, P., L. Strigini, A. Romanovsky, *Choosing Effective Methods for Design Diversity - How to Progress from Intuition to Science*, In: Proceedings of the 18th International Conference, SAFECOMP '99, Lecture Notes in Computer Science 1698, Toulouse, 1999.
10. Skiena, S., M. Revilla, *Programming Challenges*, Springer Verlag, March 2003.
11. Lee, P.A., T. Anderson, *Fault Tolerance; Principles and Practice*, Dependable Computing and Fault-Tolerant Systems, Vol. 3, Second, Revised Edition, 1981.
12. Chen, L., A. Avizienis, *N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation*, Digest of 8th Annual International Symposium on Fault Tolerant Computing, Toulouse, France, pp. 3-9, June 1978.
13. IEC, IEC61508, *Functional Safety of E/E/PE safety-related systems*, Geneva, 2001-2.

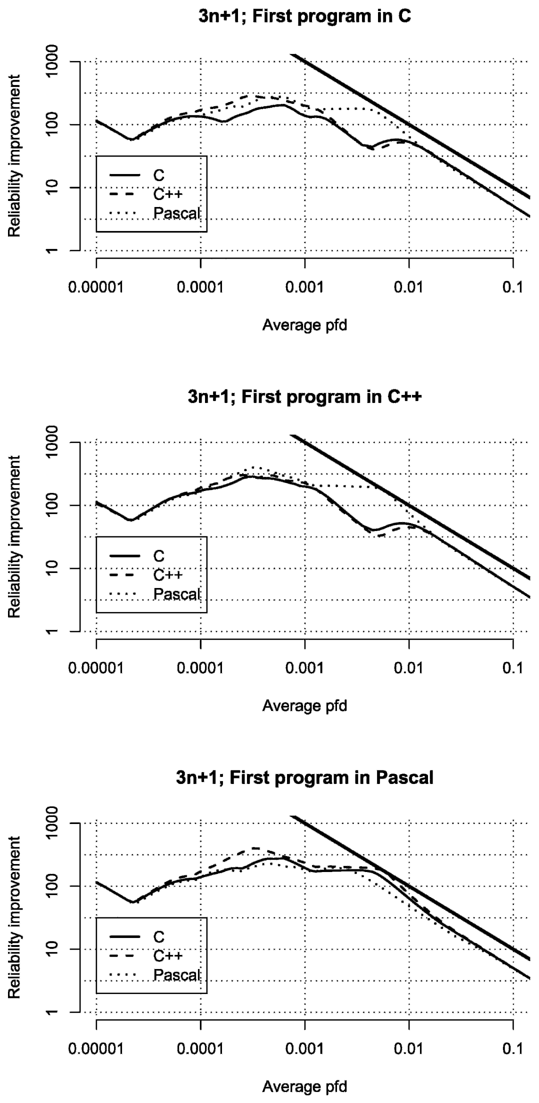

Fig. 1. These graphs show the reliability improvement of a pair of programs over a single version for the "3n+1"-problem. The horizontal axis gives the average pfd of the pools of programs involved in the calculation. In every graph, the programming language of the first program is given. The curves show the reliability improvement for the different possible choices of the programming language for the second program as function of the average pfd of the pools. The diagonal in the graphs shows the reliability achievement if the programs' behaviours were independent

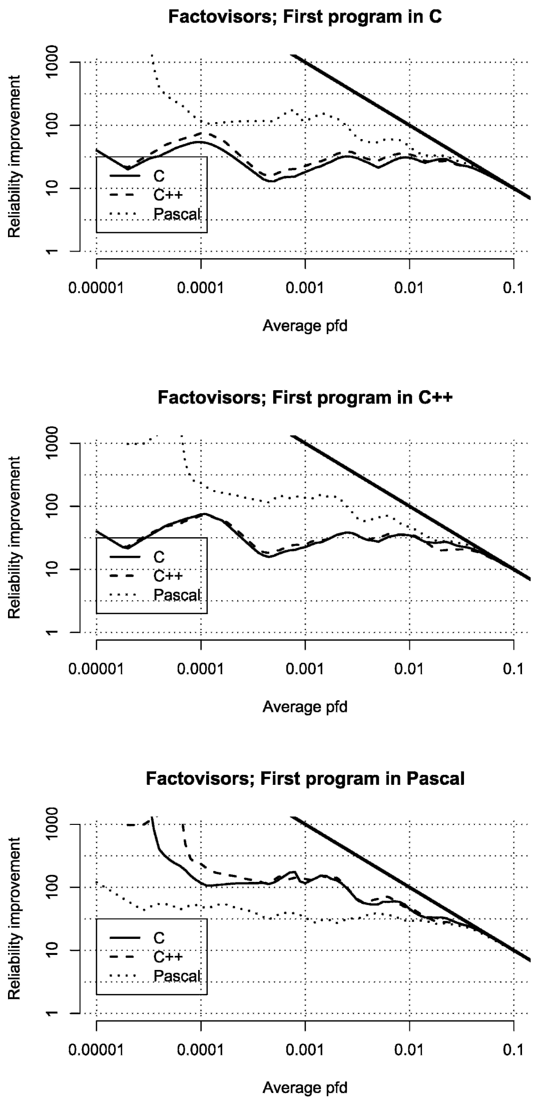

Fig. 2. The same graphs, for the "Factovisors"-problem

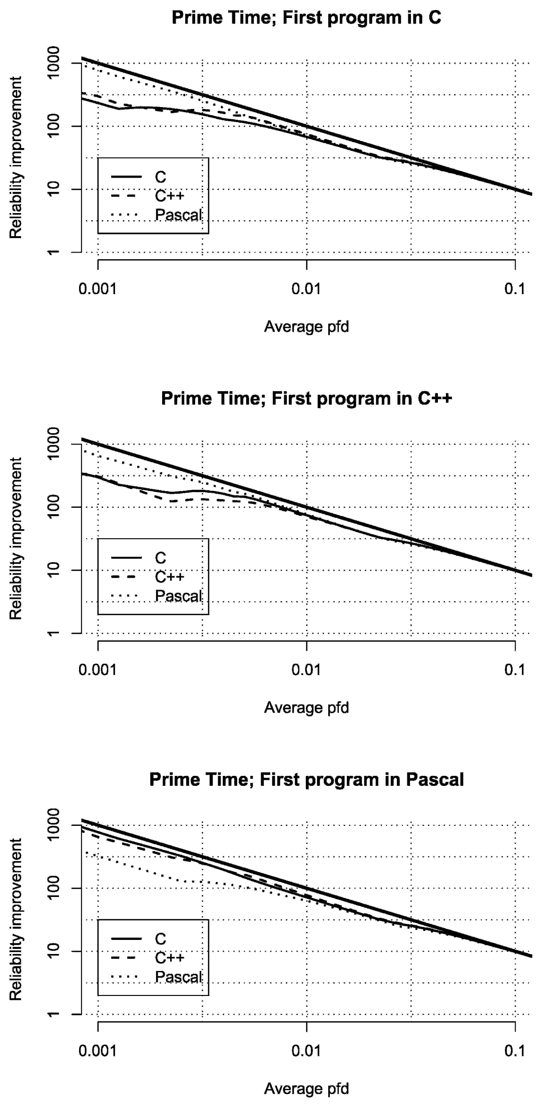

Fig. 3. The same graphs, for the "Prime Time"-problem. These graphs do not show the results for pfds below $10^{-2.5} = 3.2 \times 10^{-3}$, because the pool of Pascal programs is almost empty