Software Diversity Metrics and Measurements

Michael R. Lyu

Bellcore
Morristown, NJ 07962

Jia-Hong Chen

Bell Laboratories
Middletown, NJ 07748

Algirdas Avižienis

UCLA
Los Agneles, CA 90024

Abstract

In this paper, we define and formalize the concept of software diversity, which characterizes N-Version Software (NVS) from four different points of view that are designated as: structural diversity, fault diversity, tough-spot diversity, and failure diversity. Our goals are to find a way to quantify software diversity and to investigate the measurements which can be applied during the life cycle of NVS to gain confidence that operation will be dependable when NVS is actually employed. The versions from a six-language N-Version Programming project for fault-tolerant flight control software were used in the software diversity measurement.

1 Introduction

Fault tolerance is a function of computing systems that serves to assure the continued delivery of required services in the presence of faults which cause errors within the system [2]. We say that a unit of software (module, CSCI, etc.) is *fault-tolerant* if it can continue delivering the required services, i.e., supply the expected outputs with the expected timeliness, after *dormant* (previously undiscovered, or not removed) imperfections or "bugs", called *software faults*, have become active by producing *errors* in program flow, internal state, or results generated *within* the software unit. When the errors disrupt (alter, halt, or delay) the service expected from the software unit, we say that it has *failed* for the duration of service disruption.

An N-Version Software (NVS) unit is a fault tolerant software unit that depends on a generic *decision algorithm* to determine a *consensus result* from the results delivered by two or more ($N \geq 2$) *member versions* of the NVS unit. The *process* by which the NVS versions are produced is called *N-Version Programming* (NVP) [1]. The major objective of NVP process is to minimize the probability that two or more member versions will produce similar erroneous results that coincide in time for a decision (consensus) action of NVX [5]. This is the concept of *design diversity* [4].

The goal of design diversity is to minimize the chances of "fault leak" among independent design efforts. Furthermore, it is conjectured that the probability of a random, independent occurrence of faults that produce the same erroneous results in two or more versions is less when the versions are more diverse. A second conjecture is that even if related faults are introduced, the diversity of the member versions may cause the erroneous results not to be similar at the NVS decision. In achieving this goal, quality control of the individual software versions, using available software engineering technology and within the allowable time and cost constraints, should also be emphasized for the very simple reason that N failed versions can not produce a good result.

"Software diversity" is an attempt to describe the properties of the products of the NVP efforts, with regards to the goal of design diversity and the improvement of the qualities of the member versions. In this respect, software diversity can be specified in terms of four characteristics:

1. the structural differences among the software versions;
2. the differences between the faults found among the software versions;
3. the differences in fault-proneness among the elements of the software versions;
4. the differences in the failure behaviors among the software versions.

We will adopt the following naming scheme for the four characteristics of software diversity:

> **software diversity** has the following aspects:
>
> 1. **structural diversity**;
> 2. **fault diversity**;
> 3. **tough-spot diversity**;
> 4. **failure diversity**.

The goal of this research is to formalize the concept and notion of *software diversity* which quantifies the efficiency of *design diversity*, and to measure the NVS software diversity resulting from an NVP process. The fault-tolerant flight control software developed for the Six-Language Project [3] will be used as a case study.

The organization of the remainder of this paper is as follows: Section 2 explains the four characteristics of software diversity in more details; following that in Section 3, we show the results of applying these metrics to the programs produced during the Six-Language Project. Conclusions and future work are pointed out in the Section 4.

2 Software Diversity Metrics

2.1 Structural Diversity

Software is invisible and unvisualizable in that as soon as we attempt to depict software structure, we find it constitutes not one, but several general directed graphs superimposed one upon another [12]. Therefore, to analyze the structural differences, we would like to look at the NVS from several dimensions, perhaps further determined by the specific application. Also, there have been efforts to measure the program complexity and thereby to predict the inherent fault density. All previous efforts were done with a single version of software in mind, gathering their statistics from many programs, with most of them having different specifications [9, 8].

While one study [18, 16] has shown that these complexity measures provide little improvement over just program size alone in predicting inherent faults remaining at the start of system test, another study [11] shows that the faults found during the maintenance phase are better predicted using measures other than program size. The metric measurements were usually applied at the level of separately compilable subprograms called modules, with each module supporting one or more system functions [8]. Comparison at the level of the whole application has seldom been done in the traditional software engineering activities. The common practice is to collect the statistics from many programs which have possibly related (maybe similar, but not the same) applications in mind. It is therefore quite interesting to measure and compare the metrics at the same application level.

For NVS, we postulate that it is possible to look at and compare the individual versions of software at the subprogram (or source file) level as well as at the whole application level. We shall only try to explore some of the complexity metrics commonly seen in the literature. These basic metrics are:

- Deliverable source lines (DSL)
- Noncommentary source lines (NCSL)
- Halstead's Software Science [13]
  - Number of unique operators (η_1)
  - Number of unique operands (η_2)
  - Number of total operators (N_1)
  - Number of total operands (N_2)
- Decision count DE [8]
- McCabe's cyclomatic complexity V(G) [15]

The term "structural diversity" refers to some metrics used to compare program versions. In fact, they include both **structural metrics** and **size metrics** in the terminology of software metrics community.

2.2 Fault Diversity

The purpose of ***fault diversity*** is to demonstrate the differences between the faults introduced by the programming teams in an NVP process. For a certain set of programming teams and a given interval of the development cycle, we record and compare the faults found to determine how many kinds of faults and how many faults are detected for the set.

Def. *fault diversity* $(D_{fault}) = \frac{\textit{Number of distinct faults found}}{\textit{Total number of faults found}} = \frac{\eta_{fault}}{N_{fault}}$
(for an interval ΔT of the NVS development cycle)

The η_{fault} and N_{fault} in the above definition are similar to the ideas of η_1, η_2, N_1, and N_2 in Halstead's software science. For example, the ratio N_2/η_2 represents the average number of times operands are used. In a program where each operand is used only once, this ratio is 1. Similarly, in a group of NVS where all the faults found are different, the fault diversity is also a 1, its maximum. For the special case where no single fault is found in the set, the fault diversity is defined to be equal to 1.

Different criterion for deciding if two (or more) faults are distinct can be chosen based on which level we would like to observe the faults. It is possible to measure the fault diversity at the individual system function level of the specification and at the application level of the NVS. Furthermore, it might be interesting to observe the change of the fault diversity of the NVS as the life cycle progresses, i.e., ΔT increases.

2.3 Tough-Spot Diversity

In a large complex software project, the programmers often have difficulty with regard to certain parts of the specification. Also, it has long been agreed upon that human beings have certain blind spots when building programs [19]. Egoless programming was advocated as a partial solution to this problem.

A simple, though certainly not exhaustive, indication of the difficulties can be the percentage obtained by dividing the number of faults found in the different parts of the program by the number of faults in the entire program. Moreover, the size of the various parts of the application should be taken into account when calculating the total number of faults found. One part of the application which requires a large size of code is likely to contain more faults than the one for which a smaller size will be enough.

For NVS, a simple analysis based only on the percentage counts makes sense since we are treating each part of the application as an abstract entity and are interested in the diversity of the fault distributions among the different teams. We are curious to see what will happen when there are many teams working independently to build NVS using the same specification. If some amount of diversity of this phenomenon can be observed, it is certainly one more argument for using NVP to tolerate software design faults. Therefore, we define the "tough spots" as representing the particular system functions in a specification where a programming team has more trouble in building their software according to the specification.

There have been reports in the literature about the phenomenon of locality of faults in sections of a program [17, 20]. What the locality implies is that the probability of the existence of more faults in a section of a program is likely to be high for the section where noticeable number of faults have already been found. It is interesting to investigate this phenomenon and its implications in the context of NVS. Also, tough-spot diversity is a more hierarchical view than fault diversity and can be observed as the life cycle progresses.

2.4 Failure Diversity

When we discuss the failures of a software unit, there is always a reference to a given set Σ of input cases. Such is also the case when we define *failure diversity*, which shows the diversity in failure behaviors of a certain combination of versions. Due to diversity, failures in the components of NVS do not necessarily lead to failures of the NVS.

Def. *failure diversity* ($D_{failure}$) $= \frac{\textit{Number of distinct failures found}}{\textit{Total number of failures found}} = \frac{\eta_{failure}}{N_{failure}}|_{\Sigma}$
(This definition applies with respect to the set Σ of input cases.)

The Appendix presents some theorems leading to an intuitive result about the relationships among the failure diversity, the probabilities of failure of every software version, and the probability of failure of the corresponding NVS.

3 Software Diversity Measurements

3.1 Results of Fault Diversity

During the phases of the Six-Language Project, two pairs of common faults were found: one in the unit test phase and the other in the operation test phase. A total of 92 faults have been found so far, making the fault diversity of the six programs equal to 90/92.

The two pairs of identical faults involved four teams. It is interesting to note that both the supposed causes of the common faults were due to the specification. Unfortunately, since only two kinds of identical faults were found, we think that the information is not sufficient for further analysis of the relationships between fault diversity and other metrics of interest.

3.2 Results of Structural Diversity

By tailoring some metrics analyzers [10], special tools were written for semi-automatically measuring the basic program metrics for the six different programming languages used in the Project. This has the desirable effect that the same counting rules are applied consistently across the six programs.

To obtain the metrics at the application level, the metrics of each program's source files are added to get the application metrics. The application metrics together with the total number of faults found (d_{total}) for each program are presented in Table 1.

Notice that except for the Prolog team, the η_1 counts among the other five programs are quite close to each other. Such is not the case for η_2 counts. Figure 1 is a plot of the total number of faults found

metric	ada	c	modula	pascal	prolog	t	$\frac{max}{min}$
DSL	2256	1531	1562	2331	2228	1568	1.52
NCSL	1248	936	1069	1220	1398	1230	1.49
η_1	225	259	227	200	305	212	1.53
η_2	599	839	593	687	1294	855	2.18
N_1	3126	2722	2425	2402	3264	2111	1.55
N_2	2178	1976	1873	2047	3180	2586	1.70
DE	107	127	81	123	88	112	1.57
V(G)	150	161	121	160	181	168	1.50
d_{total}	6	18	4	12	29	23	7.25

Table 1: Comparisons of Structural Metrics at the Application Level.

after the end of all phases (d_{total}) against the number of unique operands (η_2) for each program. The linear regression line and the correlation coefficient (r) are also shown.

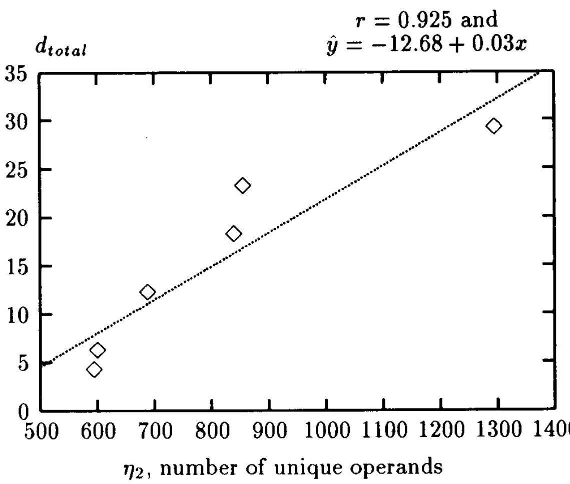

Figure 1: d_{total} vs. η_2 at the Application Level.

It is quite interesting to observe that for such a few data points, there is a strong linear relationship between these two metrics. If further demonstrated by other research, this aspect of structural diversity can help us identify fault-prone programs in an NVS life cycle.

The relationships between other metrics and d_{total}, including DSL, NCSL, η ($\eta_1 + \eta_2$), DE, and V(G), are examined in Figure 2 to Figure 6.

It should be clear that both η and V(G) have strong correlations with d_{total} (The hypothesis that η or V(G) is associated with d_{total} is accepted with a confidence level greater than 0.99). NCSL is the third, while DE and DSL perform poorly. We also define a composite metric [8] C based on η_2 and V(G): $C = \rho\eta_2 + (1-\rho)$ V(G), by varying the ρ between 0 and 1, to see if this weighted metric can perform even better. Figure 7 shows a plot of the correlation coefficients between C and d_{total} when ρ is varied.

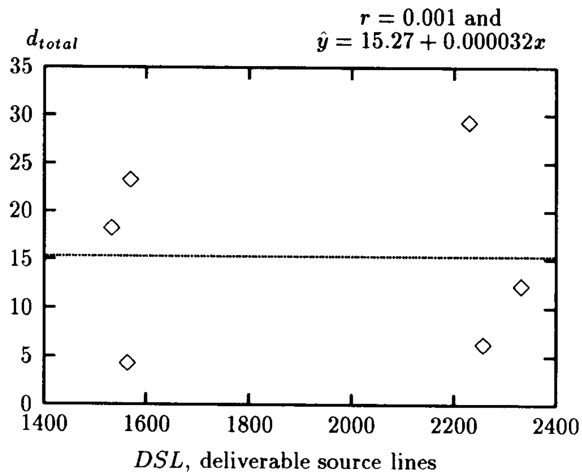

Figure 2: d_{total} vs. DSL at the Application Level.

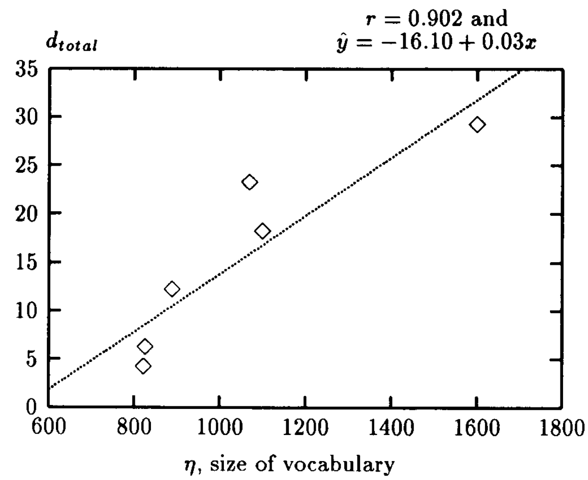

Figure 4: d_{total} vs. η at the Application Level.

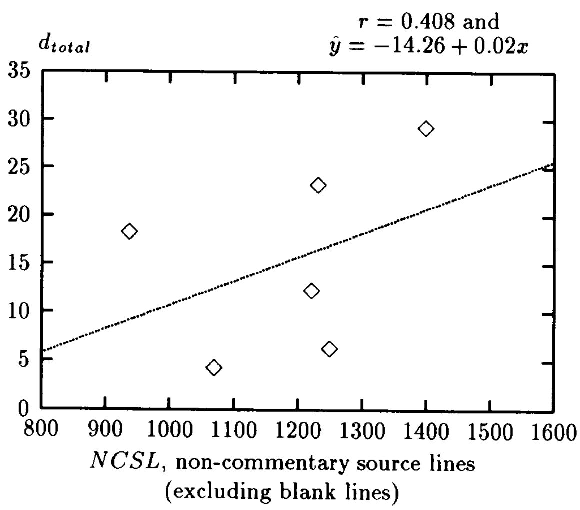

Figure 3: d_{total} vs. $NCSL$ at the Application Level.

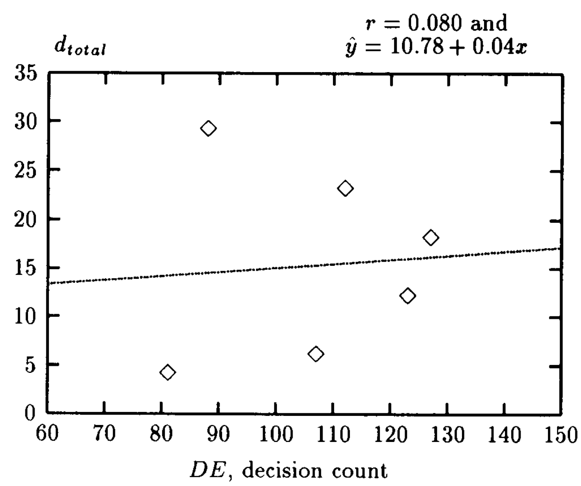

Figure 5: d_{total} vs. DE at the Application Level.

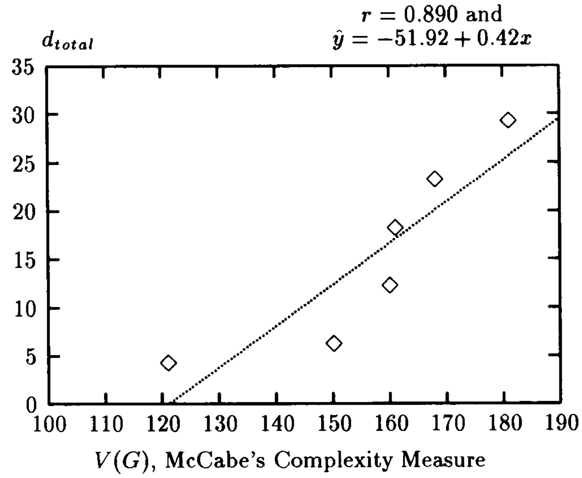

Figure 6: d_{total} vs. $V(G)$ at the Application Level.

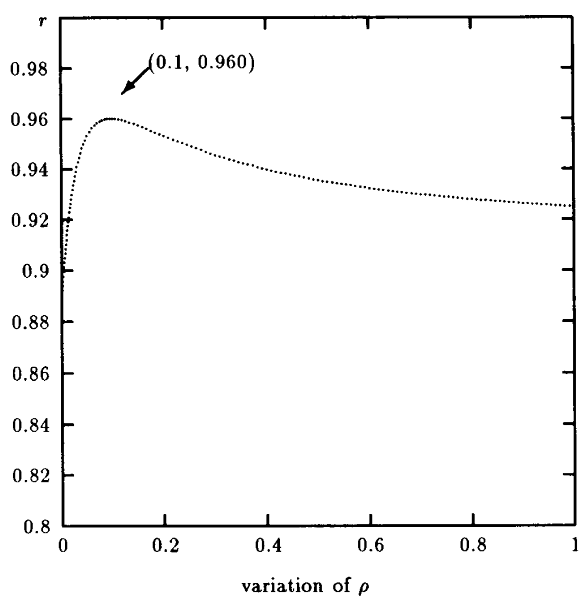

Figure 7: Correlation Coefficient r vs. Variation of ρ.

It can be seen that the correlation coefficient of C reaches its maximum when ρ is around 0.1, with an excellent r value. This combination which takes into account both the η_2 size count and V(G) logic structure complexity might be a better predictor for d_{total} for similar kinds of applications, regardless of the programming languages being used.

By going into the the source file level for each program, we can obtain similar pairs of d_{total} vs. the metrics of interest for every source file. In particular, we tried η_2, η, V(G), and NCSL. Table 2 summarizes, for each metric category and for each program, the correlation coefficients (r) with those of the application level in the last column for comparisons. The results show that the metrics measured at the source file level are not as impressive as those obtained at the application level. This implies the traditional approach of using source-file-level metrics (due to the lack of multiple occurrences of *exact* applications) to establish a predictive model for faulty density could be misleading, and as a result, inconclusive.

r	ada	c	modula	pascal	prolog	t	appl.
η_2	0.436	0.359	0.753	0.667	0.357	0.144	0.925
η	0.419	0.382	0.745	0.619	0.455	0.219	0.902
V(G)	0.159	0.113	0.644	0.299	0.579	0.032	0.890
NCSL	0.282	0.308	0.669	0.495	0.571	0.380	0.408
max(r)	η_2	η	η_2	η_2	V(G)	NCSL	η_2

Table 2: Summary of r for Each Metric Category at the Source File Level.

A complexity metric describes ***what it is, not what it has to be.*** An interesting implication of the correlations between the number of faults and structural metrics of either source files or application programs is that the structural diversity of the different levels of redundancies present in the NVS can be taken advantage of in the life cycle to concentrate our testing resources and therefore to increase the reliabilities of the corresponding parts.

3.3 Results of Tough-Spot Diversity

Let us first examine the total faults found after all the phases. Figure 8 shows six histograms which plot the percentage of faults against all the system functions (from Main to Interface) for each team.

The Ada and Modula teams' histograms are flat because in each case, there is only one fault found in the corresponding system function. It is not clear whether to call them the tough spots or not.

Table 3 summarizes the top two tough spots for each team. Whenever there is a tie for the top two all the ties are listed. The interface is not considered here. Please note that the existence of the tough spot for a team is marked by a bullet (•) in the corresponding system function. For the cases where we are not sure of the existence of a tough spot, a circle (∘) is presented.

It should be clear that ***tough-spot diversity*** also exists among the programming teams. If we summarize the top two tough spots for each team and for all six teams together before the operation test, the same results as in Table 3 are obtained (except for the Modula team) and the top two tough spots for all six teams occur in the Main, GSCF, and Inner Loop system functions.

It is interesting to note that the identical fault committed by two teams (T and Prolog) and the faults found in the C team by flight simulations during the operation test all resided in Inner Loop, which is the most common tough spot for this particular project. We postulate that the distributions of tough spots for each team and for all the teams together can guide us in allocating appropriate resources in the testing process and in the configuration of NVS system.

Function	ada	c	modula	pascal	prolog	t
Main	o				•	•
BACF	o		o			
RACF						
GSCF	o		o	•	•	
Mode Logic	o	•	o			
AH Outer						
GS Outer						
Flare Outer	o					
Inner Loop	o	•		•		•
Monitor						
Display			o	•		

Table 3: Tough Spot Distributions of Each Team.

It is postulated that there are at least three reasons which contribute to the tough-spot diversity among the programming teams:

- differences of their difficulties in grasping the concept of the application at the specification level;
- differences in their designs at the system function level (may rush forward with their first idea without giving further thoughts to other options [7]);
- differences in their implementations at the file organization level (may choose different mappings between a system function and the source files).

An intuitive hypothesis that can be made about the effect of tough-spot diversity on the failure diversity is that a larger failure diversity might be expected using the program versions which, together, have higher tough-spot diversity. The reason is that if two software versions have faults left after all the testing and during the NVS operation, but the faults reside in the different system functions of the application, the chances of these two software versions both failing on the same input case with identical results should be very rare. Even if the faults reside in the same system function, different faults are more likely to cause the system to fail in a different way.

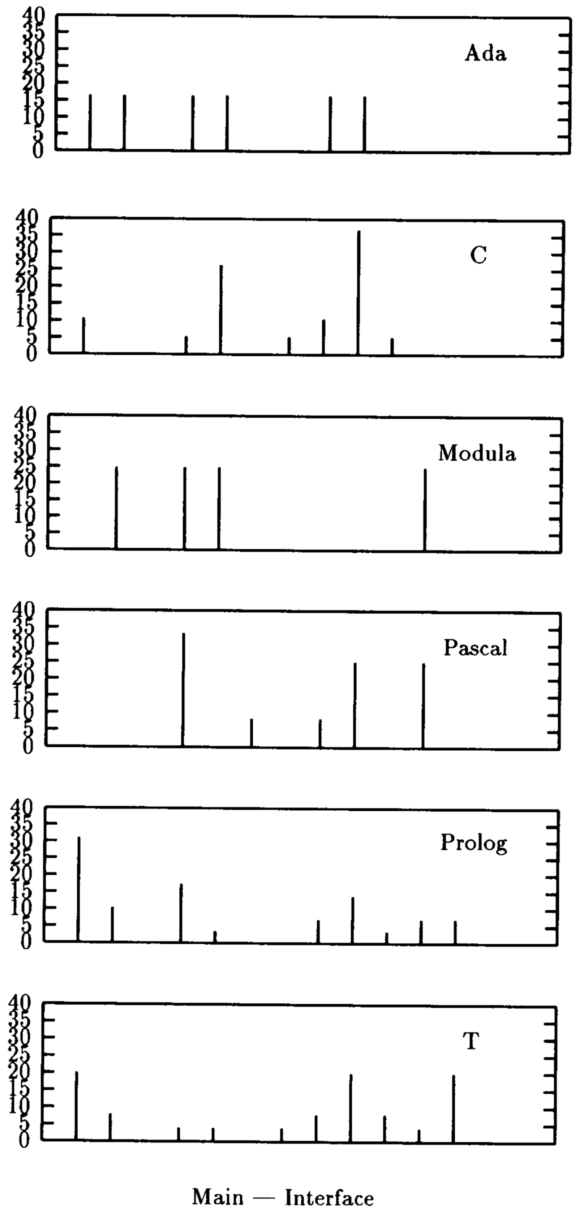

Figure 8: Histograms of the Percentage Distribution of Faults among the System Functions.

3.4 Results of Failure Diversity

Failure diversity metric is an intuitive measure of the degree by which different combinations of software versions may fail differently. After the acceptance test, many simulations matching the actual flight profile were executed during the operation test. A failure is declared if any of the intermediate or output variables deviate from those of the gold version beyond the threshold. No failures were found for the Ada, Modula, and Pascal versions. However, failures were identified for the other three versions during simulation. It was found that the Prolog and T versions had an identical fault which caused the two programs to fail identically, not counting the numerical differences introduced by the programming languages.

Identical failure is defined as two (or more) versions failing at the same time in the airplane flight path for the same input case. This is a loose criterion since the ability of NVS to mask and recover from the effects caused by faults is neglected.

In the Appendix we have established a criterion for deciding if the failure diversity for a certain NVS configuration is acceptable or not. Being acceptable means that the probability of failure of the NVS will be lower than the average probability of failure of the N versions of software.

Among the $C(6,3) = 20$ possible 3-version configurations, ten have either no failures or D (failure diversity) = 1. Among the other ten configurations, there are three equivalence classes with different D, as shown in Table 4 using the 1000 flight simulations performed so far after the acceptance test (with abbreviations A for Ada, C for C, M for Modula-2, Pa for Pascal, Pr for Prolog, T for T, and N.C. for Number of Configurations). Typically, each flight simulation takes more than 250 seconds of simulation time, with one execution through the flight control laws every 0.05 second. Thus, each successful flight requires more than 5000 executions.

Equivalence Class	D	N.C.
Pr + T + (A, M, or Pa)	0.5	3
C + (Pr or T) + (A, M, or Pa)	0.972	6
C + Pr + T	0.629	1

Table 4: 3-version Configurations with Prob. of Failure > 0.

There are $C(6,5) = 6$ possible 5-version configurations in Table 5.

Of the possible twenty 3-version configurations, there are four configurations with unacceptable diversity and probabilities of failure greater than zero. Of the six 5-version configurations, three configurations (marked with a "*") have a positive probability of failure, due to thirty identical failures found so far for the C, Prolog, and T programs. Each of these three configurations has acceptable diversity, with diversity equal to 0.629.

It is interesting to point out that among the thirty identical failures of the C, Prolog, and T programs, twenty-nine occur before 7 seconds of flight time have elapsed. One identical failure occurs at 43.30 seconds. This might suggest that for this kind of history-sensitive application which requires mainly real number computations, it is more likely that different faults will cause the versions to fail at the same time **early** in the simulation rather than **late**. Moreover, although there are thirty cases of C, T, and Prolog failing at the same time, we have not found any cases where they fail on identically the same combination of variables.

Configuration	D
A + C + M + Pa + Pr	0.972
A + C + M + Pa + T	0.972
* A + C + M + Pr + T	0.629
* A + C + Pa + Pr + T	0.629
A + M + Pa + Pr + T	0.5
* C + M + Pa + Pr + T	0.629

Table 5: 5-version Configurations.

4 Conclusions and Future Work

Software diversity is a multi-dimensional concept. Our goals in the investigations of this concept have been to first formalize it, then to study the software diversity of the Six-Language NVS products resulting from a well-defined NVP software process [1, 14, 6]. Our two major concerns, besides the assessment of the NVP products, are the intra-relationships of software diversity and the relationships between software diversity and other software attributes which can facilitate the building of NVS and increase the dependability of the final product.

Fault diversity of the six programs (90/92) is close to its maximum. No instance where a common fault occurred in more than two versions was found. While the structural diversity of the programs does not bear significant relationships with the other software diversity metrics, strong correlations between some structural metrics (η_2, $V(G)$, and $C = \rho \cdot \eta_2 + (1-\rho)V(G)$) and the number of defects found in the software at the application level have been observed. As explained before, both the failure diversity and the reliabilities of the component versions can affect the reliability of NVS. What structural diversity can provide is to indicate the potential fault ridden software component to us so that appropriate resources can be given in the NVS life cycle to improve the reliabilities of the component versions. Incidentally, the three versions which failed in the flight simulations after the acceptance test have the highest values of the three structural metrics η_2, $V(G)$, and $C = \rho \cdot \eta_2 + (1-\rho)V(G)$.

A fair degree of tough-spot diversity also exists among the six programs. Several reasons for tough-spot diversity were suggested. The faults which caused the three versions to fail in the flight simulations all resided in the Inner Loop system function, the most common tough spot among the six programs. While structural diversity has the potential to help improve the reliability of the application, tough-spot diversity might indicate possible spots (system functions) inside the application where improvements can be very beneficial.

The interplay between tough-spot (in)diversity and failure diversity was observed during the flight simulations where the tough-spot indiversity contributed to the failures of three versions at the same simulation time. From the data, it was suggested that for this application, the coincidental failures tend to happen more often in the early phase. The implication for the recovery mechanisms is that it might be more effective to spend resources in recovery in the early phase since coincidental failures caused by different faults may be the triggers.

Besides the reliabilities of the component versions, the failure diversity of the NVS is the final determining factor of the dependability of NVS. Just as in the traditional software engineering activities where we observed the growth of software reliabilities through fault removal, failure diversity of NVS will also change through time, though not necessarily growing. Study needs to be done on this aspect of diversity change through time.

Acknowledgement

The main part of this paper was written when the first author was with the ECE Department of the University of Iowa.

References

[1] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *Proceedings of COMPSAC-77*, pages 149–155, 1977.

[2] A. Avizienis and J.-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[3] A. Avizienis, M. R. Lyu, and W. Schutz. In search of effective diversity: A six-language study of fault-tolerant flight control software. In *Proceedings 18th Annual International Symposium on Fault-Toelrant Computing*, pages 15–22, Tokyo, Japan, June 1988.

[4] A. Avizienis. Design diversity — the challenge for the eighties. In *Digest of 12th Annual International Symposium on Fault-Tolerant Computing*, pages 44–45, June 1982.

[5] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[6] A. Avizienis. Software fault tolerance. In G. X. Ritter, editor, *Information Processing '89*, pages 491–498. Elsevier Science Publishers, B. V. (North Holland), 1989.

[7] J. Bentley. *Programming Pearls*. Addison-Wesley Publishing Company, 1986.

[8] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. The Benjamin/ Cummings Publishing Company, Inc., 1986.

[9] K. Christensen, G. P. Fitsos, and C. P. Smith. A perspective on software science. *IBM System Journal*, 20(4):372–387, 1981.

[10] C. R. Cook and M. Nanja. Prototype tool for managing software testing process. Technical report, Department of Computer Science, Oregon State University, Corvallis, Oregon 97331, 1986.

[11] J. S. Davis and R. J. LeBlanc. A study of the applicability of complexity measures. *IEEE Transactions on Software Engineering*, SE-14(9):1366–1372, September 1988.

[12] Jr. F. P. Brooks. No silver bullet — essence and accidents of software engineering. *IEEE Computer Magazine*, 20(4):10–19, April 1987.

[13] M. H. Halstead. *Elements of Software Science*. Elsevier North Holland Inc., New York, 1977.

[14] M. R. Lyu. *A Design Paradigm for Multi-Version Software*. PhD thesis, UCLA Computer Science Department, Los Angeles, May 1988.

[15] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[16] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Book Company, 1987.

[17] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[18] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa. Program complexity measure for software development management. In *Proc. 5th International Conference on Software Engineering*, pages 100–106, San Diego, 1981.

[19] G. M. Weinberg. *The Psychology of Computer Progarmming*. Van Nostrand Reinhold, 1971.

[20] T. J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Transactions on Software Engineering*, SE-14(9):1261–1270, September 1988.

Appendix

In the following discussion, we will simply use D, η, and N instead of $D_{failure}$, $\eta_{failure}$, and $N_{failure}$ for the purpose of conciseness.

Def. Δ = extra number of identical references for a bad version to reach majority
e.g., for three-version software (m = 3), $\Delta = 1$; for m = 5, $\Delta = 2$; in other words Δ = majority - 1, assuming m is odd, $\Delta = \frac{m-1}{2}$.

Theorem 1 *Let F_{NVS} denote the probability of failure of the NVS system:*

$$F_{NVS} \leq \frac{(1-D)}{\Delta} \times \frac{N}{|runs|}$$

Proof of Theorem 1:

Notice that $N-\eta$ is the summation of all the extra number of bad votes from all the versions of software. For a failure to occur on a certain input, the number of bad versions must constitute the majority, so that we have in general:

$$F_{NVS} \leq \frac{\frac{N-\eta}{\Delta}}{|runs|} = \frac{(1-D)\times\frac{N}{\Delta}}{|runs|} = \frac{(1-D)}{\Delta}\times\frac{N}{|runs|}$$

("=" happens when every occurrence of multiple failures is just enough to nullify the functioning of the NVS system, i.e., the number $= \Delta + 1$)

Q.E.D.

From the above theorem, it should be clear that the probability of failure of the NVS system is related to the failure diversity, the reliabilities of the individual versions, and the number of versions employed.

Theorem 2 $\frac{1}{m} \leq D \leq 1$, *where m is the number of versions employed in the NVS system and is an odd number.*

Proof of Theorem 2:

The upper bound 1 is obvious, it happens when every failure is unique in its own category, i.e., every failure occurs exactly once. The lowest value D can take on occurs when every failure replicates itself in all the other versions, i.e., it occurs m times. Every other case will have a value of D greater than $\frac{1}{m}$.

Q.E.D.

The above theorem also applies to D_{fault}, the fault diversity.

Theorem 3 *Let $m = 2\Delta + 1$, where m is defined as above, and if f_i, $1 \leq i \leq m$, represents the probability of failure of the ith version, then we have the following relationship:*

$$0 \leq F_{NVS} \leq \frac{2}{m}\cdot\sum_{i=1}^{m} f_i$$

Proof of Theorem 3:

$$F_{NVS} = \frac{1-D}{\Delta} \times \sum_{i=1}^{m} f_i, \text{ in the worst case}$$

$$\frac{1}{m} \leq D \leq 1 \Rightarrow 0 \leq (1-D) \leq \frac{m-1}{m}$$

$$\Rightarrow 0 \leq F_{NVS} \leq \frac{1-\frac{1}{m}}{\frac{m-1}{2}}\cdot\sum_{i=1}^{m} f_i \Rightarrow 0 \leq F_{NVS} \leq \frac{2}{m}\cdot\sum_{i=1}^{m} f_i$$

Q.E.D.

An intuitive explanation for this phenomenon: in order to beat the NVS system's capabilities of fault tolerance, every failure must occur in over half the number of versions. This means that when the individual probabilities of failures are summed up to determine the total system probability of failure, we have actually overestimated by half the number of versions, i.e., a factor of approximately at most $\frac{1}{\frac{m}{2}}$ for the compensation. There is a smoothing effect involved in NVS operation.

A still more accurate upper bound for F_{NVS} can be found:

Theorem 4 $F_{NVS} \leq \min(\frac{1-D}{\Delta}, D) \cdot \sum_{i=1}^{m} f_i$ **(Revised Theorem 1)**

Proof of Theorem 4:

F_{NVS} would be overestimated by **Theorem 1** in the case that it contains more than enough $(1+\Delta)$ bad versions to nullify the NVS system at the same time. A more accurate bound for this case should be:

$$F_{NVS} \leq \frac{\eta}{|runs|} = \frac{\eta}{N}\cdot\frac{N}{|runs|} = D\cdot\sum_{i=1}^{m} f_i$$

$$\text{i.e., } F_{NVS} \leq \min(\frac{1-D}{\Delta}, D)\cdot\sum_{i=1}^{m} f_i$$

Q.E.D.

Let R (risk) be the multiplication factor $\min(\frac{1-D}{\Delta}, D)$. Then a curve can be drawn of R vs. D (from $\frac{1}{m}$ to 1) for an m-version system in Figure 9.

From the curve, it is clear that for diversity greater than $\frac{m+1}{2m}$, the multiplication factor, i.e., the R value, will always be lower than the R value of a multiple-hardware-channel system running the same software, which has a D value of $\frac{1}{m}$. This leads to the following definition:

Def. An NVS system with D (failure diversity) greater than $\frac{m+1}{2m}$ is called *diversity-acceptable*. An

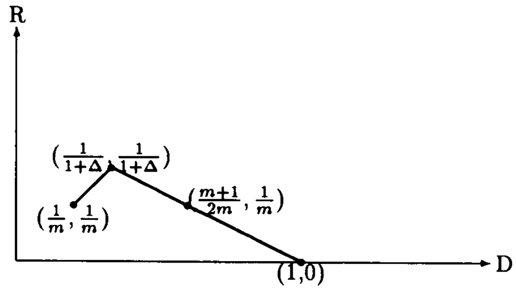

Figure 9: Relationship between Risk and Failure Diversity.

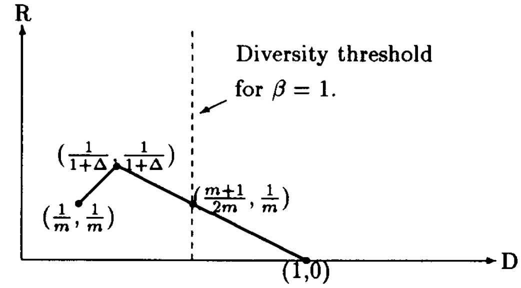

Figure 10: Relationship between Risk and Failure Diversity. (Revised)

NVS system with D not greater than $\frac{m+1}{2m}$ is ***diversity-unacceptable***.

Note that the definition of ***diversity-acceptable*** and ***diversity-unacceptable*** is only based on the multiplication factor in the formula of **Theorem 4**, i.e., the $\min(\frac{1-D}{\Delta}, D)$ in $F_{NVS} \leq \min(\frac{1-D}{\Delta}, D) \cdot \sum_{i=1}^{m} f_i$. It would be interesting to take into account the individual failure probabilities as well as the failure diversity of the NVS system.

Suppose that $f_{min} = \min(f_i, i = 1, \ldots, m)$. We want to consider the following two NVS systems:

- **NVS1**: using m identical versions with probability of failure f_{min};
- **NVS2**: using m different versions with probabilities of failure $f_i, i = 1, \ldots, m$ (assume it is ***diversity-acceptable***).

Clearly, then
$F_{NVS1} = \frac{1}{m} \cdot m f_{min}$

$F_{NVS2} = \frac{1-D}{\Delta} \cdot \sum_{i=1}^{m} f_i$, in the worst case $= \frac{1-D}{\frac{m-1}{2}} \cdot m f_{avg}$, where f_{avg} is the average probability of failure.

Now, for $F_{NVS2} \leq F_{NVS1}$

$$\Rightarrow \frac{1-D}{\frac{m-1}{2}} \cdot m f_{avg} \leq \frac{1}{m} \cdot m f_{min} \Rightarrow 1 - D \leq \frac{m-1}{2m} \frac{f_{min}}{f_{avg}}$$

$$\Rightarrow D \geq 1 - \frac{m-1}{2m} \beta, \text{ where } f_{min} = \beta f_{avg}, 0 \leq \beta \leq 1$$

For the special case when $\beta = 1 \Rightarrow D \geq 1 - \frac{m-1}{2m} = \frac{m+1}{2m}$.

Let us consider the curve of R vs. D in Figure 10, with the the added broken line representing the diversity threshold when β equals 1, i.e., when every version has the same probability of failure.

It should be clear that for β equals 1, $\frac{m+1}{2m}$ is the diversity threshold beyond which the NVS system consisting of m different versions of software pays off, but as β decreases, the threshold moves to the right. In particular, when β equals 0, the NVS system of m different software needs a diversity of 1 to be competitive. Putting aside the fact that NVS enables us to detect disagreement between versions during the operation, what this implies is that in configuring an NVS system, we should make the best effort to choose the versions with the maximum failure diversity and with as low and as compatible failure probabilities as possible.