# Analysis of coincident failing ensembles in multi-version systems

Laura Zavala and Michael N. Huhns
*University of South Carolina,*
*Computer Science and Engineering Department*
*Columbia, SC, 29208*
*{zavalagu, huhns}@engr.sc.edu*

## Abstract

*Multi-version programming is a well-known method to increase the reliability of critical software. It relies on the use multiple functionally equivalent programs or versions to provide, hopefully, a better result than that of any single version. The analysis and use of this type of system has been based on the individual reliabilities of the versions and the assumption of independence between them. However, versions might fail simultaneously and the gain from the use of diversity completely depends on the degree of dependence between the failure processes of the versions, not only on their individual reliabilities. In this paper, we present an empirical study of the correlation of code complexity measures of the versions and their coincident failures.*

## 1. Introduction

Along with the increase in pervasiveness of computer systems, there is also an increase in their complexity and the need for dependable software in both critical and everyday applications. Software fault tolerance as a strategy for achieving dependability in software is as pertinent today as it has ever been. Fault tolerance is based on the fact that it is practically impossible to produce error-free software and aims at enabling a system to continue operation even in the presence of failures or unexpected situations (e.g., erroneous inputs).

Multi-version programming, a well-known technique to achieve software fault tolerance, involves the use of multiple, independently developed and functionally equivalent, software implementations of a program (called versions). The hope is to provide, on average, a more robust behavior than that of any of the versions alone. To achieve this goal, the strengths of each version must be exploited and the weaknesses of each compensated or covered.

Some experiments have shown that common failures among the programs compromise the benefits that can be obtained with a multi-version programming approach [1], [2]. Many other experiments have reported at least some degree of improvement [13] – [17]. However, most of the experiments have been done without serious consideration of the fact that they may fail together providing identical erroneous outputs. This fact is underestimated by the use of ad hoc techniques (e.g., majority voting and weighted majority voting) for integrating contributions based on the individual reliabilities of the versions and the assumption of independence among them. These approaches do not allow for the establishment of clear design methodologies to quantify diversity and obtain the greatest possible benefit.

The gain from the use of diversity will depend on the degree of dependence between the failure processes of the versions, not just on their individual reliabilities. It has been shown theoretically [4] that "better than independence" can actually be attained. This has also been observed by experiments on diverse software fault-finding procedures [5]. Therefore, failure independence itself is clearly not the optimum result. The best effect of diversity would be a situation in which all the circumstances in which each version fails are ones where other(s) succeed, so that there is complete coverage of the space of correct solutions at any time. Consider, for example, a diverse system composed of 3 modules with reliabilities of 0.9 each. If these close-to-perfect modules exhibit high coincident failures rate (they tend to fail under the same scenarios), we might not obtain a significant increase in reliability regardless of their high individual reliabilities. Now consider the case of a diverse system composed of 3 less accurate modules with reliabilities of 0.8, 0.6, and 0.65. Suppose that every time the first one fails, the other two are correct (no matter their mediocrity). Then there is the case of perfect coverage and, if exploited adequately, the system could achieve 100% correctness.

**Table 1. The software complexity metrics used in our analyses**

| Metric | Description |
|---|---|
| Lines | Total number of lines of code |
| Statements | Total number of Statements |
| Percent Branch Statements | Ratio of lines of a branch statement to the total number of lines |
| Method Call Statements | Total number of statement that call another method |
| Percent Lines with Comments | Ratio of lines with comments to # of Code Lines |
| Classes and Interfaces | Total number of classes and interfaces |
| Methods per Class | Average number of methods in the class |
| Average Statements per Method | Average number of statements per method |
| Line Number of Most Complex Method | The number of the line where the most complex method starts |
| Maximum Complexity | Maximum Cyclomatic Complexity |
| Line Number of Deepest Block | Total number of lines in the deepest block |
| Maximum Block Depth | The maximum depth from all the nested blocks of code |
| Average Block Depth | The average depth from all the nested blocks of code |
| Average Complexity | Average Cyclomatic Complexity |
| Statements at Block Level 0 | Total number of statements that have depth 0 |
| Statements at Block Level 1 | Total number of statements that have depth 1 |
| Statements at Block Level 2 | Total number of statements that have depth 2 |
| Statements at Block Level 3 | Total number of statements that have depth 3 |
| Statements at Block Level 4 | Total number of statements that have depth 4 |
| Statements at Block Level 5 | Total number of statements that have depth 5 |

Most of the work on coincident failures has been focused on their prevention and usually consists of qualitative guidance on how to enforce diversity deliberatively during the design and development phases (different programming languages, different programmers' backgrounds, etc.). Some empirical models have also been developed to detect common faults in programs, which can lead to coincident failures. All these approaches are intended to be used prior to release and operation of a system.

We are interested in the problem of how to configure a redundant system during operation, so that the design space is optimally explored and the highest failure coverage possible, as allowed by the diversity of the modules, is attained. This should be complementary and not alternative to techniques for enforcing diversity during design and development. Of course, when significant independence in the variants' failure processes can be achieved, a simple adjudicator can be used, and multi-version programming provides effective error recovery from design faults. It is likely, however, that completely independent development cannot be achieved in practice [18].

In this paper we present our empirical studies on the correlation of code complexity measures for each group and their coincident failures. At an abstract level, this can be viewed as the correlation of structural diversity and functional diversity. Our work should be considered as a first step to study the use of code complexity measures as an indirect way of representing dependences among diverse versions and, thus, provide estimates of proneness to coincident failures. This in fact can be applied in both pre-operation and during operation of a multi-version system. In the first case, it can provide additional information to the estimation of diversity among programs. In the second case, it can provide heuristic information to the adjudicator for deciding on a final result.

## 2. Related work

A few works have provided models for the study of coincident failures between diverse programs. Voas [3] and Dai [6] have proposed testing of the $2^N - 1$ ensembles into which N versions can be decomposed. Voas [3] presents an algorithm and a software analysis prototype to observe common-mode failures produced by combinations of simulated programmer faults. They simulate faults (through fault injection techniques) for every ensemble from the $2^N$-1 ensembles and keep count of the coincident failures observed for each. The goal is to be able to predict how the software will behave if real faults exist in the multiple versions. Dai [6] presents a model of correlated failures in logically exclusive CCF (Common Cause Failures) events, with which the reliability function of the dependent N-version programming can be easily derived using fault-tree analysis. They also decompose the failure space

**Table 2. The operations that each doubly linked list program in our experiment implements. An input string specifies a sequence of operations to be applied to the list. Each character in the string corresponds to one of the operations that can be performed on the list.**

| Operation | Description | Char |
|---|---|---|
| insert (Object newElement) | Inserts newElement after the cursor. If the list is empty, then newElement is inserted as the first (and only) element in the list. In either case, moves the cursor to newElement. The element to insert is the character following the '+' in the input string. | + |
| remove ( ) | Removes the element marked by the cursor from the list. Moves the cursor to the next element in the list. Assumes that the first list element "follows" the last list element. | - |
| replace (Object newElement) | Replaces the element marked by the cursor with newElement and leaves the cursor at newElement. The element to replace the current element is the character following the '=' in the input string. | = |
| clear ( ) | Removes all elements in the list. | C |
| gotoBeginning ( ) | If the list is not empty, moves cursor to beginning of the list and returns true, else returns fales. | < |
| gotoEnd ( ) | If the list is not empty, moves cursor to end of the list and returns true, else returns false. | > |
| gotoNext ( ) | If cursor not at end of the list, moves cursor to next element in the list and returns true else returns false. | N |
| gotoPrior ( ) | If cursor not at beginning of the list, moves cursor to preceding element in the list and returns true, else returns false. | P |
| getCursor ( ) | Returns the element at the cursor. | @ |

into $2^N$-1 ensembles (which they refer to as components).

Some approaches associate static code metrics with defects, but only for individual programs. Basili [9] presented an experiment with eight student teams where they found that object oriented (OO) metrics appeared to be useful for predicting defect density. A survey on empirical studies showing that OO metrics are significantly associated with defects can be found in Subramanyam and Krishnan [10]. Hudepohl [11] successfully predicted whether a module would be defect prone or not by combining metrics and historical data. Nagappan [8] presented an experimental study of the correlation between software complexity measures and post-release observed failures in large scale systems. They built regression models that accurately predict the likelihood of post-release defects for new entities.

Finally, some have explored the use of software metrics to detect program plagiarism. Several models have been proposed to measure program similarity [7]. However the similarity measured for plagiarism is of different nature that the similarity/diversity studied in multi-version systems.

## 3. Design study

### 3.1. Some definitions

**3.1.1. Faults and failures.** Faults are flaws in a system which can be caused by different reasons such as incorrect specification or an incorrect implementation. Failures are the consequences of encountering the faults during operation or execution of the system. Failures are observable errors in the program behavior. In other words, every failure can be traced back to some fault, but not every fault will result in a failure.

In a multi-version system, a coincident failure occurs when two or more versions of a program are identically incorrect. Coincident failures do not have to be caused by identical faults in the versions; however, that is the predominant cause, and such failures are referred to in the literature as common mode failures (CMF) or common cause failures (CCF).

**3.1.2. Software metrics.** A software metric is a measure of some property of a piece of software or its specifications. The software complexity metrics that we used for our analyses are briefly explained in Table 1.

**3.1.3. Ensembles of programs.** In this paper we use the term ensemble (of programs) to refer to one of the possible groupings that can occur from a pool of *N* programs. The total number of ensembles that can be obtained from *N* programs is given by:

$$\sum_{j=1}^{N} \binom{N}{j} = 2^N - 1$$

**3.1.4. Adjudicators.** Adjudication is the process where an output is computed based on the results provided by the diverse versions in a multi-version system. Simple majority voting has been the predominant mechanism used for adjudication. Other voting strategies that have also been explored include consensus or plurality, maximum likelihood, and weighted voting. The latter two make use of the individual reliabilities of the versions. All of these

approaches rely heavily on an assumption of independence among the versions. Of course, when significant independence in the variants' failure processes can be achieved, a simple adjudicator can be used. However, the independence in the variants' failure processes is usually assumed.

### 3.2 Contributions

Our work constitutes a first step towards combining failure history of ensembles of versions that exhibit coincident failures with other parameters, specifically, code complexity measures. The hope is that we can determine if the latest can be used as an estimator of proneness of the ensembles to coincident failures (we believe this must hold for at least some multi-version systems). At an abstract level, this can be viewed as the combination of observed structural diversity with observed functional diversity.

For the moment, we only address the basic question of whether software complexity metrics correlate with observed coincident failures. Our work should be considered an initial effort towards the establishment of formal methodologies for the use of code complexity measures as an indirect way of representing dependences among diverse versions, and thus, provide estimates of proneness to coincident failures. Interesting related questions, which we cannot generalize for the moment, are: do programs that behave similarly contain the same kind of faults? Do programs that behave similarly have similar static metrics? Can software metrics help in determining programs similarity, and thus, proneness to coincident failures?

If we can determine, for particular instances of multi-version software systems, that there is at least a set of software metrics that provides useful information (besides observed failures) about possible dependences between the versions, then we can use that information for several purposes. In a pre-operational phase, for example, it can be used as an indicator to the estimation of diversity among programs and thus help in design decisions (e.g. the selection/exclusion of the versions to include in the system). In an operational phase, it can be used as heuristic information for deciding the versions to use for a particular run, or as heuristic information to the adjudicator for deciding on a final result. The adjudicator could, for example, Confidence levels could be assigned to the different outputs given by coalitions of programs that coincide. This would be different than the weights usually assigned based on the individual reliabilities (past performance) of the versions.

### 3.3 Experimental setup

**3.3.1 The programs.** We collected one set of 28 Java implementations of an algorithm for performing series of sequential operations on a doubly linked list. Different people wrote each program. In this case, the class structure (i.e., method signatures) was specified, so the differences among the algorithms are in performance and correctness.

Each algorithm maintains a doubly linked list and a pointer to the current element on the list (cursor). The functionality of each algorithm can be summarized as follows: 1) Initialize the list to empty; 2) Read input string; 3) Apply to the list the sequence of operations specified on the input string; 4) Return the resulting list. Each character in the input string corresponds to one of the operations that can be performed on the list. Therefore, the input string corresponds to a sequence of operations to be applied to the list. The allowed operations and their corresponding character are presented in Table 2.

**3.3.2 Failure data.** For models that explore all the possible ensembles, such as those presented in [3], [6], a combinatorial problem arises. They claim that the total is not intrinsically too large since $N$ is usually a small odd integer, such as three, five or seven, for practical implementations of multi-version systems.

We used the models proposed by Voas [3] and Dai [6] as a basis for maintaining record of the coincident failures. However, we do not perform testing on the $2^N - 1$ ensembles into which the $N$ versions can be decomposed. In our case, their claim does not hold – the design space is not small ($2^{28}$). Instead, we execute all the versions for each test case and keep record only of the ensembles that do exhibit coincident failures. This is usually a much smaller set.

A set of 1000 input strings, representing sequences of operations, were randomly created. For each input case, we invoke the 28 programs, providing them the corresponding input string. The results of each program are analyzed and the programs giving the same incorrect output are grouped together. The coincident failures count is increased by one for each of these ensembles.

After performing 1000 input cases on the 28 programs, we have a coincident failure count of each ensemble of programs from the $2^N - 1$ ensembles into which the $N$ versions can be decomposed.

**Table 3. Correlation of code complexity measures of the versions (average and variance) and their coincident failures.**

| Metric | Average per Ensemble | Variance per Ensemble |
|---|---|---|
| Lines | -0.14342 | -0.01868 |
| Statements | -0.238 | **-0.44286** |
| Percent Branch Statements | 0.261434 | **-0.41978** |
| Method Call Statements | 0.311124 | **-0.42528** |
| Percent Lines with Comments | 0.004517 | -0.18791 |
| Classes and Interfaces | -0.17307 | 0.072527 |
| Methods per Class | 0.374082 | **-0.51648** |
| Average Statements per Method | **0.51157** | -0.15714 |
| Line Number of Most Complex Method | -0.07002 | 0.053846 |
| Maximum Complexity | 0.234049 | **-0.44835** |
| Line Number of Deepest Block | 0.133258 | 0.172527 |
| Maximum Block Depth | 0.248447 | -0.16044 |
| Average Block Depth | 0.35799 | **-0.44066** |
| Average Complexity | **0.40880** | **-0.51758** |
| Statements at block level 0 | -0.17307 | 0.072527 |
| Statements at block level 1 | -0.29164 | -0.23077 |
| Statements at block level 2 | -0.32693 | **-0.42088** |
| Statements at block level 3 | **0.58865** | -0.06044 |
| Statements at block level 4 | 0.089215 | **-0.46484** |
| Statements at block level 5 | 0.248447 | -0.17033 |

**3.3.3 Metrics data.** For each one of the 28 programs, we calculate their corresponding value for each of the metrics listed in Table 1. For space reasons, we do not show such values here. We then use the individual measures of the programs to calculate, for each of the ensembles of programs exhibiting coincident failure behavior, the average and the variance of such measures.

**3.3.4 Calculating the correlation.** We determined the correlation between the averages and variances complexity measures of each ensemble with the number of coincident failures exhibited by that ensemble. For this purpose, we use the Spearman rank correlation [12], which is a commonly used and robust correlation technique because it can be applied even when the association between elements is non linear.

The Spearman rank correlation has been used in previous software reliability experiments. For example, Nagappan [8] used it in their experiments for calculating the correlation between software complexity measures and post-release observed failures in large scale systems (single systems, as opposed to multi-version systems).

## 4. Results

The resulting standard Spearman correlation coefficients are shown in Table 3. The coefficients represent how well the average and variance complexity of the programs in the ensembles exhibiting coincident failures, correlate with the number of coincident failures observed. Highlighted values indicate significant correlation. *Average per Ensemble* is the correlation with number of coincident failures in ensembles based on average metric values. *Variance per Ensemble* is the correlation with number of coincident failures in ensembles based on variance metric values.

As it can be observed from Table 3 the *average complexity* and the *average number of statements per method* of the programs in an ensemble have a positive correlation to the number of coincident failures exhibited by that ensemble. This is analogous to results of previous studies on individual programs, showing that software metrics can be estimators of software faults [8-11].

On the other hand, we can observe the negative correlation between the variance of several metrics of the programs in an ensemble and the number of coincident failures of the ensemble. This maybe interpreted as if the structural similarity, in our particular experiment setup, of programs in the ensembles positively correlates to failure proneness. Likewise, the variance (structural diversity) negatively correlates to failure proneness.

## 5. Discussion and Conclusion

In the same way that software metrics have been used as estimators of faults in individual programs, we have proposed their use in multi-version systems as possible estimators of coincident failures. Analogous to results of previous studies on individual programs showing that software metrics can be estimators of software faults [8-11], we have presented experimental evidence showing a correlation between software metrics and coincident errors. As of the current state of our study we can not yet generalize. The programs used for our experiments are more representative of programming in the small than large commercial software. Also, there was no methodology followed for the development of the programs, nor for assuring diversity. We need more empirical analyses before we can generalize our observations. However, we have shown that the use of software complexity metrics as indicators of the proneness to coincident failures of multi-version systems is worth exploring.

Of course, there is no such thing as a magic or

golden method to predict failures in software, much less, coincident failures in multi-version systems. Neither is there a single set of metrics that fits all projects. However, it seems that, for some projects, it might be possible to find a set of complexity metrics that correlates with coincident failures. This could be exploited, for example, when selecting the versions to include in a system, or the versions to use for a particular run. It can also be used as heuristics for the adjudication strategy to decide on the final result.

We have provided only a particular situation where, for a multi-version system, there is a correlation between software metrics and coincident failures and we plan to further extend our empirical studies.

## 6. References


[1] J.C. Knight and N.G. Leveson, An experimental evaluation of the assumption of independence in multi-version programming. IEEE Trans. Software Engineering 12 (1986).

[2] S.S. Brilliant, J.C. Knight and N.G. Leveson, Analysis of Faults in an N-Version Software Experiment, IEEE Trans. Software Engineering 16, 2 (1990), 238-247.

[3] Voas, J., Ghosh, A., Charron, F., and Kassab, L., Reducing Uncertainty About Common-Mode Failures. In Proceed of the 8th Intl Symposium on Software Reliability Engineering (ISSRE '97), IEEE Computer Society, Washington, DC, 308.

[4] Littlewood, B. and Miller, D. R., Conceptual Modeling of Coincident Failures in Multiversion Software. IEEE Trans. Software Engineering 15, 12 (Dec. 1989), 1596-1614.

[5] Littlewood, B., Popov, P. T., Strigini, L., and Shryane, N., Modeling the Effects of Combining Diverse Software Fault Detection Techniques. IEEE Trans. Soft. Eng. 26, 12 (Dec. 2000), 1157-1167.

[6] Y. S. Dai, M. Xie, K. L. Poh, S. H. Ng, A model for correlated failures in N-version programming, IIE Transactions 36, 12 (2004), 1183-1192.

[7] Whale, G. 1990. Software metrics and plagiarism detection. J. Syst. Softw. 13, 2 (Oct. 1990), 131-138.

[8] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In Proceed of the 28th Intl Conference on Software Engineering (ICSE '06). ACM, New York, NY, 452-461.

[9] V. R. Basili, L. C. Briand, and W. L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering, 22(10), 1996.

[10] R. Subramanyam and M. S. Krishnan, Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, IEEE Transactions on Software Engineering, 29(4) pp. 297-310, April 2003.

[11] J. P. Hudepohl, Aud, S.J., Khoshgoftaar, T.M., Allen, E.B., Mayrand, J., Emerald: software metrics and models on the desktop, IEEE Software, 13(5), pp. 56 - 60, 1996.

[12] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach: Brooks Cole, 1998.

[13] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. Computer, August 1984.

[14] J.R. Parker., Voting methods for multiple autonomous agents. In Proceed of the 3rd Australian and New Zealand Conference on Intelligent Inf. Systems, Australia, 1995.

[15] P. Townend and J. Xu. Assessing multi-version systems through fault injection. In Proceed of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), Computer Society.

[16] P. Townend, P. Groth, and J. Xu. A provenance-aware weighted fault tolerance scheme for service-based applications. In ISORC '05: Proceedings of the 8th IEEE Intl Symposium on Object-Oriented Real-Time Distributed Computing, 2005. IEEE Computer Society.

[17] Rosa Laura Zavala Gutierrez and Michael N Huhns. On building robust web service-based applications. In Lawrence Cavedon and others, eds, Extending Web Services Technologies: The Use of Multi-Agent Approaches, Chapter 14, Kluwer Academic Publishing, New York, 2004.

[18] M Donnelly, B Everett, J Musa, G Wilson, Best Current Practice of SRE, Handbook of Software Reliability Engineering, M. Lyu, Ed., Mc Graw, 1996.