NASA Contractor Report 187492

Software Reliability Experiments Data Analysis and Investigation

J. Leslie Walker and Alper K. Caglayan

Charles River Analytics Inc.
Cambridge, MA 02139

The Charles Stark Draper Laboratory Inc.
Cambridge, MA 02139

Contract NAS1-18061

January 1991

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This report describes the research done on CSDL Subcontract No. 791. The program versions and failure data used in this research were produced under NASA Contract number NAS1-17705. This previous work is referred to as the *RSDIMU experiment*. The main thrust of the current work is to: 1) develop an understanding of the fundamental reasons making redundant software components fail dependently, and 2) investigate the construction of software fault tolerant structures maximizing the independence between developed software components.

The work described in this report falls into four categories:

1. Error Analysis
2. Acceptance Check Development
3. N-Version vs. Recovery Block Analysis
4. Analysis of Software Diversity

Under error analysis, we examine the particular errors made by programmers in the RSDIMU experiment in order to determine what the effect of programmer errors is on the failure behavior of redundant software components. We are particularly interested in errors which are not similar in nature, but cause coincident failures. This analysis is important in identifying software development methodologies to improve the independence of programmer errors which cause coincident failures. This analysis is also important in determining the limitation of independent software development (i.e. what degree of independent failure behavior we can expect even if we are able to achieve independence between the errors made by the authors of different redundant software components).

Under acceptance check development, we develop the theory of a generalized acceptance check development methodology, and present examples of applications of this theory. This methodology will produce acceptance checks which fail independently from the software whose correctness they are designed to check. This work is important in building recovery block structures. In addition, this work is applicable to the validation of high reliability software.

Under N-Version vs. Recovery Block analysis we have constructed models which predict

the reliability of these two types of fault tolerant software structures. These models predict the performance of such software structures depending upon the number of versions implemented, and the reliability of the associated acceptance check. Using these models we compare these two approaches to fault tolerant software development in order to determine which is appropriate under certain conditions. In particular, we compare these models using the programs from the RSDIMU experiment as a realistic example of a redundant component population and determine under what conditions we should choose N-Version programming and when to choose recovery blocks in order to most efficiently use the available resources.

Under our analysis of software diversity, we examine the effects of diverse methodologies on the development of redundant software components. We are particularly interested in the impact of enforced diversity on reliability gain, and in development methodologies that produce independent failure behavior. There are two results that we have achieved with this analysis: 1) models for predicting the reliability of diverse software structures, and 2) an understanding of what type of diversity is required. We obtain these results both in the general case and in our particular examination of the results of the RSDIMU experiment.

2. ANALYSIS OF SOFTWARE ERRORS IN RSDIMU PROGRAMS

In this section we present our analysis of the software errors which were identified in each of the 20 RSDIMU programs and the effects of these software errors on the failure behavior of the programs. Specifically, we describe the software errors which occur in the programs and what misconceptions of the programmers caused them to make these errors. We also present a comparison between the correlation of failures between separate programs and the similarities between the errors made by the programs' authors. Finally, we group the programs according to development and certification sites and examine the failures and the errors in order to determine the effects of site diversity.

We relate software errors to each other as shown in figure 2.1. A *conceptual error* results from the programmer's misconception about how the program is ultimately meant to behave. When the programmer implements this misconceived program, their conceptual errors are manifested as logical errors. A logical error is the particular error which causes a program to fail. We often treat logical errors independently of the misconceptions which caused them.

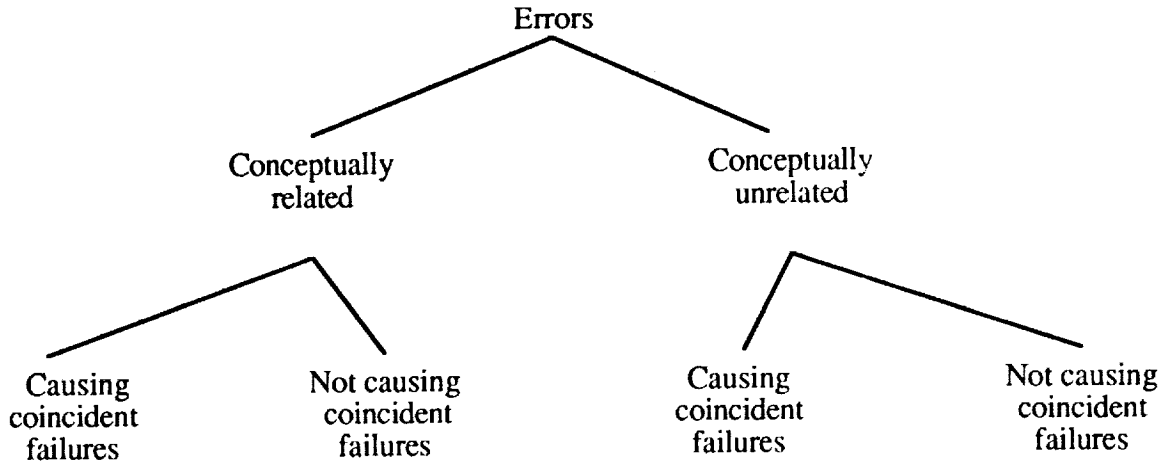

Figure 2.1: Categories of Relationships between Logical Errors.

Two logical errors are said to be *conceptually related* if they result from the same or similar misconceptions on the part of the respective programmers. We further relate errors by whether or not they tend to cause their respective programs to fail together. We consider each logical error to have an input domain. This domain is the set of all inputs for which a particular logical error will

cause a failure. If the input domains of two logical errors overlap significantly then we say that these logical errors cause *coincident failures*.

2.1. Software Error Descriptions

In this section we describe three additional conceptual software errors which were not described in volume 3 of R8903. Two of these additional conceptual errors caused failures in only the U2 ($S_{0,1}$) case and these are described in section 2.1.1. The remaining conceptual error is described in section 2.1.2. The combination of this section and volume 3 of R8903 describe all of the errors which we analyzed in this study. A summary of the conceptual errors that we examined in detail is shown in table 2.1.1.

The inputs to the RSMIMU programs are divided into 6 groups based on the parameters used to generate the input data. These parameters govern the difficulty with which the programs must come up with a correct answer for the data. The names used to refer to these input cases are:

$U1(S_{0,0})$, $U2(S_{0,1})$, $U3(S_{1,0})$, $U4(S_{1,1})$, $U5(S_{2,0})$, and $U6(S_{2,1})$.

2.1.1. Failures in case U2 ($S_{0,1}$)

In this section we describe the errors which were discovered in the U2 case ($S_{0,1}$). These errors all occurred in the sensor failure isolation routines of the RSDIMU programs. Inputs for the U2 case often created situations where an edge relation was violated which was common to two "good" faces. This caused two situations which were not properly handled by many of the programmers:

1. Three edge relations failed, but there was no face common to them.
2. Four edge relations failed.

The first of these situations was not correctly handled by versions uclac, uclad, and uvac. The second of these situations was not correctly handled by versions uclac, uclad, uiuca, uiuce, and uvac. In general, the versions failed because they assumed that these situations would never occur. In some cases, however, some of the programs still managed to give the correct outputs even though the algorithm that was used would only give the correct outputs in special situations.

Fault Number	Software Faults	Versions
1	A unit vector in an orthogonal coordinate system was apparently assumed to remain a unit vector after a nonorthogonal transformation	ncsud uclae
2	Failure isolation algorithm was implemented in a coordinate system other than specified	uclac uiucd uvac
3	Vector components were apparently assumed to remain the same after a small angle transformation	uvab
4	Three edge out-of-tolerance edge relations were apparently assumed to have a face common to all of the out-of-tolerance edge relations	uclac uclad uvac
5	Four out-of-tolerance edge relations were not processed	uclac uclad uiuca uiuce uvac
6	Test threshold computed incorrectly	uiuca
7	Variable initialized incorrectly	uclab
8	Best estimate of acceleration left uninitialized after system failure	uiuca uclab

Table 2.1.1: Summary of Conceptual Errors in RSDIMU Versions.

Version uclac failed under both of the situations described. The authors assumed that if more than one edge relation failed for more than one face, then the system should fail. This is true if more than one edge relation fails for more than *three* faces and not two. The result was that when four edge relations were violated, a system failure was indicated. In some cases three violated edge relations also triggered a system failure due to the same fault.

Version uclad also failed under both of the situations described. The failure isolation algorithm contained a case statement whose tag was the number of failed edge relations. The authors grouped the 4, 5, and 6 edge relation out-of-tolerance cases together and failed the system in all of these situations. This caused the program to fail in all cases where 4 edge relations were

out of tolerance. Version uclad failed whenever three edge relations were out of tolerance with no face common to them. The authors specifically detected this situation and failed the system, which was incorrect.

Version uiuca failed only when 4 edge relations were out-of-tolerance. The authors use the total number of times an edge relation is out-of-tolerance for each face -- thus each time an edge relation is out-of-tolerance it counts twice -- to determine whether or not a failure has occurred. When any three edge relations are out-of-tolerance, this total is 6 and the case is handled directly. However, when 4 edge relations are out-of-tolerance the total is 8 and the authors did not consider this possibility. In such a case no effort was made to isolate a sensor failure on any of the faces, and the version failed.

Version uiuce also failed only when 4 edge relations were out-of-tolerance. The authors made independent tests on every combination of 3 from 6 edge relations that could be out-of-tolerance with a common face between them. The algorithm is to check the edge relation involved with face A and each of faces B, C, and D. The first one of these edge relations that is out-of-tolerance is assumed to be the only possibility for failure. Thus if the AB edge relation is out-of-tolerance, then only the BC,BD and AC,AD edge relations are checked. If four edge relations are out-of-tolerance with a face common to three of them and AB is not one of these three, then this version will assume that no face has failed the edge-vector test. This causes failures only in 3 of the possible 15 cases where four edge relations can fail, so version uiuce did not fail in all of the four edge out-of-tolerance cases.

The authors of version uvac assume that whenever three edge vector relations are out-of-tolerance there will be a face common to them. They then assigned a failure signature to each of the 4 possible combinations of edge relation which could be out-of-tolerance with a single face in common. This signature can be computed directly from the edge relations and examined to see which face fails. Since there are actually 15 possible combinations of 3 edge relations this algorithm does not always work. However when a signature does not match one of the 4 key values, no face is assumed to have failed. This produces the correct result for most of the three edge relation combinations. The program fails because combinations where there is no common face to the three edge relations will produce a signature that indicates the failure of one face. Since this only occurs for 2 of 15 possible combinations of three edge relations, the program did not fail in all of the cases of three edge relations out-of-tolerance with no common face.

Version uvac also contains a case statement similar to that of version uclad where the tag value is the number of out-of-tolerance edge relations. Once again, the 4, 5, and 6 edge relation

cases are grouped together. Since system failure detection is handled prior to this point in the program, nothing is done for these cases and no failure is detected. The prior system failure detection algorithm does not handle the case of 4 edge relations out-of-tolerance and so in this case no failure is detected.

2.1.2. Failure to Properly Indicate Estimation Status

We diagnosed an error in version uclab which was not described in report R8903. This error causes failures whenever a system failure is signaled due to lack of good faces for an analytical estimate of acceleration. The specification instructs programmers to set the estimate status of each of the 5 estimates (best estimate and each channel estimate) to UNDEFINED whenever a system failure is signaled. The authors of version uclab set the channel estimate status to UNDEFINED in this case, but allow the best estimate to remain uninitialized. Thus the status of the best estimate remains at whatever value is set by the compiler for initialization. This often causes failures when the program fails the system.

Section 2 of R8903 describes an error in version uiuca which is similar to that of uclab. When a system error is signaled the value of the best estimate is not set to zero and the status is not set to UNDEFINED. The effects of this error are identical to that of the uclab best estimate error.

2.2. Analysis of Errors and Failures in RSDIMU Programs

Here we present our analysis of the logical errors which occur in the 20 RSDIMU versions. We examine the correlation of these errors and how this relates to the conceptual and input domain relationships between the logical errors.

Table 2.2.1 shows the correlation of the occurrence of the conceptual errors of table 2.1.1. The numbers along the diagonal of this grid (x_{AA}) represent the number of times a conceptual error occurs in version A. The numbers off the diagonal (x_{AB}) represent the number of conceptually-related errors which occur between versions A and B.

Table 2.2.2 shows the input correlation of failures between the RSDIMU programs. The numbers along the diagonal (x_{AA}) represent the total number of inputs on which version A failed. The numbers off the diagonal (x_{AB}) represent the total number of inputs on which both version A and B failed together. It is difficult to judge the significance of the numbers since they represent

only the count of failures. In order to dtermine whether or not two versions exhibit significantly correlated failure behavior we performed a χ^2 test with the null hypothesis:

H_0 = "Versions A and B exhibit independent failure behavior"

for each combination of versions A,B following the same statistical test as [Brilliant, Knight, and Leveson 1989].

	ncsua	ncsub	ncsuc	ncsud	ncsue	uclaa	uclab	uclac	uclad	uclae	uiuca	uiucb	uiucc	uiucd	uiuce	uvaa	uvab	uvac	uvad	uvae
ncsua	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ncsub	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ncsuc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ncsud	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
ncsue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uclaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uclab	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0
uclac	0	0	0	0	0	0	0	3	2	0	1	0	0	1	1	0	0	3	0	0
uclad	0	0	0	0	0	0	0	2	2	0	1	0	0	0	1	0	0	2	0	0
uclae	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
uiuca	0	0	0	0	0	0	1	1	1	0	3	0	0	0	1	0	0	1	0	0
uiucb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uiucc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uiucd	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0
uiuce	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	1	0	0
uvaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uvab	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
uvac	0	0	0	0	0	0	0	3	2	0	1	0	0	1	1	0	0	3	0	0
uvad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uvae	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.2.1: Correlated Occurrence of Conceptual Errors in RSDIMU Program

	ncsua	ncsub	ncsuc	ncsud	ncsue	uclaa	uclab	uclac	uclad	uclae	uiuca	uiucb	uiucc	uiucd	uiuce	uvaa	uvab	uvac	uvad	uvae
ncsua	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ncsub	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ncsuc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ncsud	0	0	0	129	0	0	20	66	0	67	49	0	0	62	0	0	66	66	0	0
ncsue	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
uclaa	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uclab	0	0	0	20	0	0	53505	18	0	27	20923	0	0	14	0	0	19	22	0	0
uclac	0	0	0	66	1	0	18	995	466	88	224	0	1	100	48	0	108	352	0	0
uclad	0	0	0	0	1	0	0	466	466	0	185	0	1	0	48	0	0	247	0	0
uclae	0	0	0	67	0	0	27	88	0	180	68	0	0	88	0	0	89	87	0	0
uiuca	0	0	0	49	0	0	20923	224	185	68	42043	0	0	36	48	0	40	223	0	0
uiucb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uiucc	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
uiucd	0	0	0	62	0	0	14	100	0	88	36	0	0	101	0	0	101	100	0	0
uiuce	0	0	0	0	0	0	0	48	48	0	48	0	0	0	48	0	0	48	0	0
uvaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uvab	0	0	0	66	0	0	19	108	0	89	40	0	0	101	0	0	110	105	0	0
uvac	0	0	0	66	0	0	22	352	247	87	223	0	0	100	48	0	105	362	0	0
uvad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
uvae	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.2.2: Input Correlation of Failures between RSDIMU Programs.

The χ^2 statistic can be computed from table 2.2.2 using:

$$\chi^2_{AB} = \frac{(x_{AB} - p_A p_B n)^2}{p_A p_B n} + \frac{((x_{AA} - x_{AB}) - p_A(1-p_B)n)^2}{p_A(1-p_B)n} + \frac{((x_{BB} - x_{AB}) - (1-p_A)p_B n)^2}{(1-p_A)p_B n} + \frac{((n-(x_{AA} + x_{BB} - x_{AB})) - (1-p_A)(1-p_B)n)^2}{(1-p_A)(1-p_B)n} \quad 2.1.$$

where p_A is the average reliability of version A. For each pair of versions A,B we compare this statistic with a threshold of $\alpha = 5\%$ level with 3 degrees of freedom. The result of this analysis is table 2.2.3. A dot "." in this table represents comparisons which are meaningless because our testing was unable to produce failures in either or both of the versions in question. A star "*" in this table represents a comparison in which the statistic was not large enough to cause rejection of the null hypothesis of independence. In this case the versions are assumed to fail independently. Finally, a "C" in this table represents a comparison in which the statistic was large enough to cause rejection of the null hypothesis. In this case the programs were assumed not to have failed independently. In order to determine whether the programs exhibited dependent or better than independent failure behavior we computed the correlation coefficient for the two failure behaviors. A "-C" indicates that the programs exhibited better than independent failure behavior.

Each of the conceptual errors in table 2.1.1 can be traced to a particular logical error in each program involved. We have determined the number of failures in each program due to each logical error which was found to occur in the program from table 2.1.1. The correlation between these individual logical errors is shown in table 2.2.4. We applied the same χ^2 test to these correlations to obtain a correlation shown in table 2.2.5. By comparing tables 2.2.3 and 2.2.5 to table 2.2.1 we observe that some of the failure correlation between versions can be attributed to the existence of logical errors which are conceptually-related. Where table 2.2.1 indicates that two versions have one or more conceptually-related logical errors in common, presumably these related logical errors are the cause of the correlation in table 2.2.3. The following pairs of versions have

significant failure correlation which can be attributed to conceptually-related logical errors:

(uclae,ncsud)	(uclab,uiuca)	(uclac,uclad)
(uclac,uiuca)	(uclac,uiucd)	(uclac,uiuce)
(uclac,uvac)	(uclad,uiuca)	(uclad,uiuce)
(uclad,uvac)	(uiuca,uiuce)	(uiuca,uvac)
(uiucd,uvac)	(uiuce,uvac)	

	ncsua	ncsub	ncsuc	ncsud	ncsue	uclaa	uclab	uclac	uclad	uclae	uiuca	uiucb	uiucc	uiucd	uiuce	uvaa	uvab	uvac	uvad	uvae
ncsua	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
ncsub	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
ncsuc	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
ncsud	.	.	.	C	*	*	C	C	*	C	C	.	*	C	*	.	C	C	.	.
ncsue	.	.	.	*	C	*	*	C	C	*	*	.	C	*	*	.	*	*	.	.
uclaa	.	.	.	*	*	C	*	*	*	*	*	.	*	*	*	.	*	*	.	.
uclab	.	.	.	C	*	*	C	-C	-C	C	C	.	*	C	*	.	C	*	.	.
uclac	.	.	.	C	C	*	-C	C	C	C	C	.	C	C	C	.	C	C	.	.
uclad	.	.	.	*	C	*	-C	C	C	*	C	.	C	*	C	.	*	C	.	.
uclae	.	.	.	C	*	*	C	C	*	C	C	.	*	C	*	.	C	C	.	.
uiuca	.	.	.	C	*	*	C	C	C	C	C	.	*	C	C	.	C	C	.	.
uiucb	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
uiucc	.	.	.	*	C	*	*	C	C	*	*	.	C	*	*	.	*	*	.	.
uiucd	.	.	.	C	*	*	C	C	*	C	C	.	*	C	*	.	C	C	.	.
uiuce	.	.	.	*	*	*	*	C	C	*	C	.	*	*	C	.	*	C	.	.
uvaa	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
uvab	.	.	.	C	*	*	C	C	*	C	C	.	*	C	*	.	C	C	.	.
uvac	.	.	.	C	*	*	*	C	C	C	C	.	*	C	C	.	C	C	.	.
uvad	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
uvae	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Table 2.2.3: Significance of Correlated Occurrence of Failures in RSDIMU Programs.

		ncsud	uclab		uclac			uclad		uclae	uiuca			uiucd	uiuce	uvab	uvac		
		ncsud.1	uclab.7	uclab.8	uclac.4	uclac.5	uclac.2	uclad.4	uclad.5	uclae.1	uiuca.5	uiuca.6	uiuca.8	uiucd.2	uiuce.5	uvab.3	uvac.4	uvac.5	uvac.2
ncsud	ncsud.1	121	26	0	0	0	64	0	0	68	0	54	0	60	0	64	0	0	64
uclab	uclab.7	26	49327	0	0	0	17	0	0	31	0	71	18488	16	0	17	0	0	21
	uclab.8	0	0	2197	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1
uclac	uclac.4	0	0	0	411	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uclac.5	0	0	0	0	438	0	244	191	0	180	0	0	0	47	0	47	180	0
	uclac.2	64	17	1	0	0	104	0	0	83	0	41	0	98	0	104	0	0	102
uclad	uclad.4	0	0	0	0	244	0	244	0	0	0	0	0	0	0	0	47	0	0
	uclad.5	0	0	0	0	191	0	0	191	0	180	0	0	0	47	0	0	180	0
uclae	uclae.1	68	31	1	0	0	83	0	0	185	0	77	0	83	0	83	0	0	82
uiuca	uiuca.5	0	0	0	0	180	0	0	180	0	181	0	0	0	47	0	0	181	0
	uiuca.6	54	71	1	0	0	41	0	0	77	0	314	0	38	0	41	0	0	40
	uiuca.8	0	18488	0	0	0	0	0	0	0	0	0	41528	0	0	0	0	0	0
uiucd	uiucd.2	60	16	1	0	0	98	0	0	83	0	38	0	99	0	98	0	0	98
uiuce	uiuce.5	0	0	0	0	47	0	0	47	0	47	0	0	0	47	0	0	47	0
uvab	uvab.3	64	17	1	0	0	104	0	0	83	0	41	0	98	0	105	0	0	102
uvac	uvac.4	0	0	0	0	47	0	47	0	0	0	0	0	0	0	0	47	0	0
	uvac.5	0	0	0	0	180	0	0	180	0	181	0	0	0	47	0	0	181	0
	uvac.2	64	21	1	0	0	102	0	0	82	0	40	0	98	0	102	0	0	112

Table 2.2.4: Input Correlation Between Occurrences of Logical Errors in RSDIMU Programs

		ncsud	uclab		uclac			uclad		uclae	uiuca			uiucd	uiucc	uvab	uvac		
		ncsud.1	uclab.7	uclab.8	uclac.4	uclac.5	uclac.2	uclad.4	uclad.5	uclae.1	uiuca.5	uiuca.6	uiuca.8	uiucd.2	uiucc.5	uvab.3	uvac.4	uvac.5	uvac.2
ncsud	ncsud.1	C	C	*	*	*	C	*	*	C	*	C	*	C	*	C	*	*	C
uclab	uclab.7	C	C	-C	-C	-C	C	-C	-C	C	-C	C	C	C	*	C	*	-C	C
	uclab.8	*	-C	C	*	*	*	*	*	*	*	*	-C	*	*	*	*	*	*
uclac	uclac.4	*	-C	*	C	*	*	*	*	*	*	*	-C	*	*	*	*	*	*
	uclac.5	*	-C	*	*	C	*	C	C	*	C	*	-C	*	C	*	C	C	*
	uclac.2	C	C	*	*	*	C	*	*	C	*	C	*	C	*	C	*	*	C
uclad	uclad.4	*	-C	*	*	C	*	C	*	*	*	*	-C	*	*	*	C	*	*
	uclad.5	*	-C	*	*	C	*	*	C	*	C	*	-C	*	C	*	*	C	*
uclae	uclae.1	C	C	*	*	*	C	*	*	C	*	C	-C	C	*	C	*	*	C
uiuca	uiuca.5	*	-C	*	*	C	*	*	C	*	C	*	-C	*	C	*	*	C	*
	uiuca.6	C	C	*	*	*	C	*	*	C	*	C	-C	C	*	C	*	*	C
	uiuca.8	*	C	-C	-C	-C	*	-C	-C	-C	-C	-C	C	*	*	*	*	-C	*
uiucd	uiucd.2	C	C	*	*	*	C	*	*	C	*	C	*	C	*	C	*	*	C
uiucc	uiucc.5	*	*	*	*	C	*	*	C	*	C	*	*	*	C	*	*	C	*
uvab	uvab.3	C	C	*	*	*	C	*	*	C	*	C	*	C	*	C	*	*	C
uvac	uvac.4	*	*	*	*	C	*	C	*	*	*	*	*	*	*	*	C	*	*
	uvac.5	*	-C	*	*	C	*	*	C	*	C	*	-C	*	C	*	*	C	*
	uvac.2	C	C	*	*	*	C	*	*	C	*	C	*	C	*	C	*	*	C

Table 2.2.5: Significance of Correlation Between Failures due to Logical Errors.

There are still quite a few pairs of failure correlated versions in table 2.2.3 which cannot be attributed to conceptually related errors. For example, errors 1, 2, 3, and 6 of table 2.1.1 have input domains which will overlap in cases where the misalignment angles are large and/or the tolerance parameter NSIGT is small (indicating very little tolerance between the edge vectors). The following pairs of version have errors which are conceptually-unrelated, but cause coincident failures:

(ncsud,uclac)	(ncsud,uiuca)	(ncsud,uiucd)
(ncsud,uvab)	(ncsud,uvac)	(uclac,uclae)
(uclac,uvab)	(uclae,uiuca)	(uclae,uiucd)
(uclae,uvab)	(uclae,uvac)	(uiuca,uiucd)
(uiuca,uvab)	(uiucd,uvab)	(uvab,uvac)
(uclab,uclac)	(uclab,uclad)	

The existence of conceptually-related logical errors is one factor that leads to correlated failure behavior. It is possible that certain conceptual errors are caused by the development methodologies. Diverse development methodologies may reduce the failure correlation of this type.

The existence of overlapping input domains is another factor that leads to correlated failure behavior. The existence of a logical error in a version indicates that the input domain for that logical error lies outside the domain of inputs that was included by the testing methodology. Identical testing methodologies as well as methodologies whose input domains overlap are more likely to allow logical errors with overlapping input domains. Diverse testing methodologies may reduce the occurrence of this type of related errors.

Errors 7 and 8 are shown by table 2.2.5 to produce negative correlation with most of the other conceptual errors. Most significantly, logical error uclab.7 causes negative correlation with versions uclac and uclad through errors uclac.4, uclac.5, uclad.4, and uclad.5. These errors are conceptually unrelated.

2.3. Analysis of Errors and Failures in Development and Certification Site Subpopulations

Diversity of development methodologies is one way to inhibit the occurrence of logical

errors which cause coincident failures. Here we examine the diversity which exists in the RSDIMU experiment in order to determine the degree to which this diversity minimizes the occurrence of logical errors which cause coincident failures. The RSDIMU programs were developed by independent development teams at four development sites. They were then certified at three different certification sites. The different sites lead to the determination of two different diverse classifications of programs. These are classification according to development site and classification according to certification site. Table 2.3.1 shows the correlation of the occurrence of conceptually-related logical errors broken into groups by development site. Table 2.3.2 shows the correlation of the occurrence of conceptually-related logical errors broken into groups by certification site. By totalling the number of conceptually-related logical errors between each pair of versions at each pair of sites, we obtain the data for tables 2.3.3 and 2.3.4. These are a measure of how well the diversity avoided the occurrence of conceptually-related logical errors. We then performed a χ^2 test with the null hypothesis:

H_0 = "The occurrence of conceptually-related errors in programs produced by sites A and B was independent"

For two different sites A and B in the following tables, the rejection of H_o indicates that these sites produce programs which contained a significantly large number of conceptually related errors. In this case we conclude that the diversity between these sites is unsuccessful in reducing the occurrence of conceptually related errors. The acceptance of the H_o indicates that this type of site diversity might be a good method of reducing the occurrence of conceptually related errors.

The results of this test are shown in tables 2.3.5 and 2.3.6. Here we see that the only site which produced versions in which the occurrence of conceptually-related logical errors is not correlated is the development site ncsu. This information seems to contradict the failure correlation information of table 2.2.3 in which development sites ncsu and uva show a relatively small amount of failure correlation.

		ncsu					ucla					uiuc					uva				
		ncsua	ncsub	ncsuc	ncsud	ncsue	uclaa	uclab	uclac	uclad	uclae	uiuca	uiucb	uiucc	uiucd	uiuce	uvaa	uvab	uvac	uvad	uvae
ncsu	ncsua	0	0	0	.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ncsub	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ncsuc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ncsud	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	ncsue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ucla	uclaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uclab	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0
	uclac	0	0	0	0	0	0	0	3	2	0	1	0	0	1	1	0	0	3	0	0
	uclad	0	0	0	0	0	0	0	2	2	0	1	0	0	0	1	0	0	2	0	0
	uclae	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
uiuc	uiuca	0	0	0	0	0	0	1	1	1	0	3	0	0	0	1	0	0	1	0	0
	uiucb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uiucc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uiucd	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0
	uiuce	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	1	0	0
uva	uvaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uvab	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	uvac	0	0	0	0	0	0	0	3	2	0	1	0	0	1	1	0	0	3	0	0
	uvad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uvae	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.3.1: Correlated Occurrence of Conceptual Errors in RSDIMU Programs by Development Site

		ncsu							ucsb						uva						
		ncsuc	ncsud	ncsue	uclad	uiucc	uvaa	uvac	uclac	uclae	uiuca	uiucb	uiucd	uvab	ncsua	ncsub	uclaa	uclab	uiuce	uvad	uvae
ncsu	ncsuc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ncsud	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	ncsue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uclad	0	0	0	2	0	0	2	2	0	1	0	0	0	0	0	0	0	1	0	0
	uiucc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uvaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uvac	0	0	0	2	0	0	3	3	0	1	0	1	0	0	0	0	0	1	0	0
ucsb	uclac	0	0	0	2	0	0	3	3	0	1	0	1	0	0	0	0	0	1	0	0
	uclae	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	uiuca	0	0	0	1	0	0	1	1	0	3	0	0	0	0	0	0	1	1	0	0
	uiucb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uiucd	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
	uvab	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
uva	ncsua	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ncsub	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uclaa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uclab	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	2	0	0	0
	uiuce	0	0	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0
	uvad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	uvae	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.3.2: Correlated Occurrence of Conceptual Errors in RSDIMU Programs by Certification Site.

	ncsu	ucla	uiuc	uva
ncsu	-	1	0	0
ucla	1	-	6	5
uiuc	0	6	-	3
uva	0	5	3	-

Table 2.3.3: Correlation of Occurrence of Conceptual Errors Between Development Sites.

	ncsu	ucsb	uva
ncsu	-	9	2
ucsb	9	-	3
uva	2	3	-

Table 2.3.4: Correlation of Occurrence of Conceptual Errors Between Certification Sites.

	ncsu	ucla	uiuc	uva
ncsu	-	*	*	*
ucla	*	-	C	C
uiuc	*	C	-	C
uva	*	C	C	-

Table 2.3.5: Significance of Correlation of Occurrence of Conceptual Errors Between Development Sites.

	ncsu	ucsb	uva
ncsu	-	C	C
ucsb	C	-	C
uva	C	C	-

Table 2.3.6: Significance of Correlation of Occurrence of Conceptual Errors Between Certification Sites.

The reason for the apparent contradiction is the invalidity of the assumption that conceptually-related errors will always cause coincident failures and that conceptually-unrelated errors will not cause coincident failures. Table 2.3.7 shows the failure intensity distributions corresponding to each of the conceptual errors of table 2.1.1. For each conceptual error e_i the

corresponding failure intensity distribution is compiled using only the failures which were attributable to logical errors which were in turn attributable to the conceptual error e_i. Table 2.3.8 shows the corresponding version failure distribution for each e_i.

An example of conceptually-related errors which do not cause coincident failures is uclad.4 with uclac.4 and uvac.4. From the descriptions of the logical errors attributable to conceptual error number 4 in section 2.1.1 we expect that the input domain of uclac.4 would entirely contain the input domain of uclad.4 which would entirely contain the input domain of uvac.4. Table 2.3.7, however, shows that there are no 3 version coincident failures attributable to conceptual error number 4. This means that a significant part of the input domains of these logical errors does not overlap.

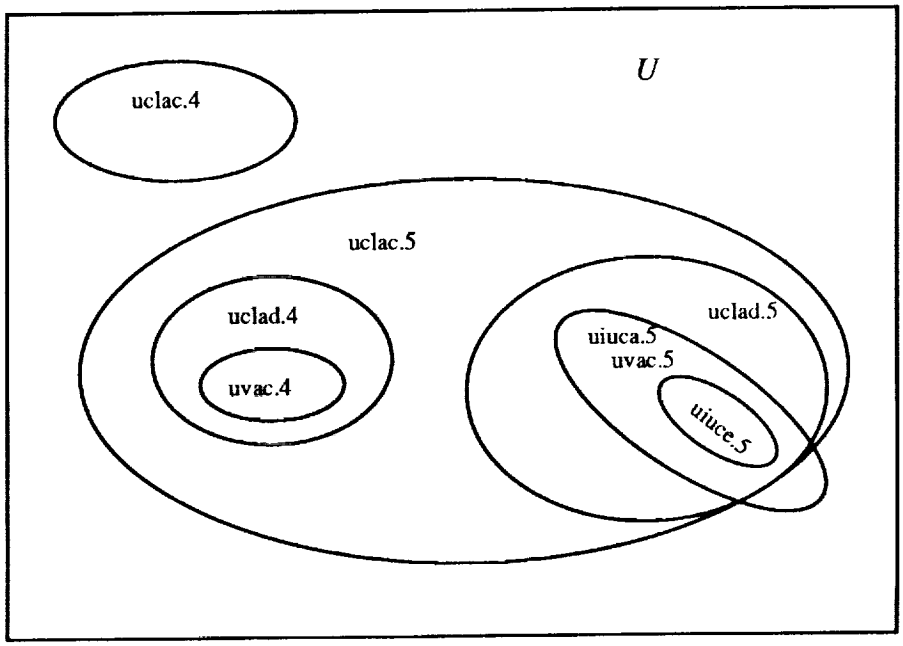

Figure 2.3.1: Venn Diagram of Input Domains of Conceptual Errors 4 and 5.

Detailed analysis of the input domains of logical errors attributable to conceptual errors 4 and 5 leads to the diagram of figure 2.3.1. This is a Venn diagram of the input domains for each of these logical errors. One interesting feature of this diagram is that logical errors uiuca.5 and uvac.5 have the same input domain which strongly supports the assertion that conceptually-related errors cause coincident failures. Notice also, however, that the domain of the failures of uclac.4 lie completely outside of the other failures attributable to conceptual error number 4. Further, the input domain of uclad.4 and uvac.4 lie entirely within the input domain of uclac.5. This indicates

that the failures of uclad.4 and uvac.4 are more correlated with uclac.5 which is non-related than with uclac.4 which is conceptually-related.

Another example of conceptually-related errors that do not cause coincident failures is uclab.8 with uiuca.8. Notice that logical errors uclab.8 and uiuca.8 are conceptually-related and yet the input domain of these errors is disjoint (Table 2.2.4).

An example of a version containing conceptually-unrelated errors which cause coincident failures is uclab. We have determined that version uclab contains logical errors attributable to conceptual errors 7 and 8, and that it is the only version containing logical errors attributable to conceptual error number 7. This means that version uclab contains logical errors which are conceptually unrelated with any versions but uiuca and that the input domains of uclab.8 and

	Error Number							
	1	2	3	4	5	6	7	8
1	170	13	105	608	247	314	49323	43725
2	68	4	0	47	12	0	0	0
3	0	98	0	0	0	0	0	0
4	0	0	0	0	133	0	0	0
5	0	0	0	0	47	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0

Table 2.3.7: Failure Intensity Distribution for Conceptual Errors.

	Error Number							
	1	2	3	4	5	6	7	8
ncsud	121	0	0	0	0	0	0	0
uclab	0	0	0	0	0	0	49323	2197
uclac	0	104	0	411	438	0	0	0
uclad	0	0	0	244	191	0	0	0
uclae	185	0	0	0	0	0	0	0
uiuca	0	0	0	0	181	314	0	41528
uiucd	0	99	0	0	0	0	0	0
uiuce	0	0	0	0	47	0	0	0
uvab	0	0	105	0	0	0	0	0
uvac	0	112	0	47	181	0	0	0

Table 2.3.8: Version Failure Distributions for Conceptual Errors.

uiuca.8 are disjoint. In spite of this apparent independence of version uclab, table 2.2.3 shows version uclab to exhibit correlated failure behavior with 6 other versions. In addition table 2.2.5 shows uclac.7 to be failure correlated with 8 conceptually-unrelated errors.

These observations lead us to believe that the correlation of conceptually-related logical errors is highly dependent upon the type of conceptual error and the manifestation of this error. The nondeterministic nature of an uninitialized variable makes the correlation of logical errors resulting from conceptual error number 7 difficult to predict. The design of version uclab is such that the effects of uclab. 7 tend to cause correlation with unrelated errors.

The manifestation of logical errors uclab.8 and uiuca.8 are such that the erroneous code is called upon in situations whose input domains are disjoint. This behavior is not determined by conceptual error number 8 alone. Further, it cannot be determined by logical errors uclab.8 and uiuca.8 alone. In order to determine the effects of conceptual error number 8 upon the failure behavior of versions uclab and uiuca, it is necessary to understand the computations of these programs in detail.

Finally, the nature of conceptual error number 5 is such that the input domains of logical errors attributable to it significantly overlap with each other. This leads us to believe that the nature of conceptual error number 5 is completely different from that of 7 or 8.

2.4. Results of Error Analysis

Conceptually-related logical errors cause coincident failures. Diversity of development methods and diversity of testing methods may help to inhibit the occurrence of coincident failures caused by conceptually-related logical errors. However, the elimination of conceptually related errors is not enough since conceptually unrelated errors can cause dependent failure behavior as well. Here, the interaction between a logical error and the computations of the program and the interaction between different logical errors with overlapping input domains is important. Enforcement of diversity upon development and testing methodologies needs to take into account the degree to which a methodology inhibits or promotes these types of interaction.

The occurrence of unrelated logical errors causes negative correlation. The enforcement of diversity upon development and testing methodologies should promote the production of programs whose errors are conceptually- and input-unrelated. In addition, relationships other than input-related and conceptually-related should be developed in order to relate logical errors which will cause coincident failures and logical errors which will not.

We have found that site diversity is not enough of a factor in development to significantly reduce the occurrence of conceptually-related errors. Since conceptually related errors cause coincident failures, site diversity is not likely to reduce the numbers of coincident failures in the development of multiple versions.

3. ACCEPTANCE CHECK DEVELOPMENT

In this section we present our theory of generalized interactive program checkers. In addition we present some examples of complex programs for which we have developed interactive checkers.

3.1 Interactive Proofs

Interactive Proofs is a new approach to software validation. Interactive Proofs is different than formal verification in that each particular output for a given input is "verified" in contrast to verifying the correctness of software for all inputs. Here, we give an informal overview of this approach to checking of program correctness. This approach owes much to the development of the theory of Interactive Proofs. [Blum and Raghavan 1988] [Babai 1989] [Goldwasser, Micali, and Rackoff 1985]

The scenario we are considering below is as follows: consider a program P supposedly satisfying specification S, and producing the output P(x) on some input x. We would like to know how much trust we can have in the correctness of this particular output. The approach taken here is different from program correctness proving techniques in that it is much more practical in general, though it tells us only if the program is correct in the particular instances. The problems for which efficient program checkers were constructed include examples from graphs, codes, matrices, latin squares, and multisets.

The approach is to construct a program checker C which checks whether the output P(x) of a program P on input x is indeed the answer S(x) on input x, where P is a program for S. The checker C depends on the specification S and not on the program P. Thus modifications of P do not require changes in C. The checker C is a probabilistic algorithm. Program P produces an error on input x if $P(x) \neq S(x)$. In such a case the checker will output INCORRECT with an overwhelming probability, depending only on a security parameter k. It is very important to stress that the probabilities (of checker's error, for example) are over the internal *coin tosses* of C, and depend on neither program P nor inputs x. If $P(x) = S(x)$, i.e. no error occurs in the particular instance, the checker C may still output INCORRECT if there is another instance x', such that $P(x') \neq S(x')$. If the program P is always correct the checker will always output CORRECT. It is

natural to require that the checker spends less time than the program. In some cases it is even possible to construct checkers which run much faster than the program they check.

Example 1: Graph Isomorphism. Let program P given two graphs produce 1 if the graphs are isomorphic and 0 otherwise. Let $P(G_0,G_1) = 0$ for graphs G_0 and G_1. The verification of this output may be very hard since there is no way known to produce a short proof of graph non-isomorphism. But the following checker will correctly identify an error in $P(G_0, G_1) = 0$ with probability $\frac{1}{2^k}$.

```
DO k times
    i := random_bit (ie. i ∈ {0,1})
   T := random_permutation (Gi)
    if P(T,G0) = i then return BUGGY
END DO
return CORRECT
```

Note that the checker uses the specification knowledge that graph isomorphism is invariant under random permutations. If the checker returns BUGGY, then the bug can be localized since P was wrong either in $P(G_0, G_1)$ or in $P(T,G_0$, assuming that T has been correctly computed).

We would like to stress interactive proof simplicity of this checker's algorithm. Much of the theory so far has been developed for NP decision type problems. In the next section, we discuss its generalization to arbitrary programs.

3.2 Generalized Interactive Checkers

Blum and Raghavan define a test procedure for asserting the correctness of a program solving a decision/search problem in NP on a given input. Here, we discuss one possible generalization to testing arbitrary programs by reducing to a decision problem. Let

X be the set of all possible inputs for a problem
Y be the set of all possible outputs for a problem

Define the program specification S by:

$S: X \rightarrow Y$ such that $S(x) = y$ for $x \in X, y \in Y$

Let P be a program supposedly satisfying the specification, then

$P: X \rightarrow Y$ such that $P(x) = y$ for $x \in X, y \in Y$

Let x by an arbitrary input on which the program P produces the output y. We would like to assert whether y is correct or not. We define two equivalence classes. Now, there are two *natural* equivalence relations that we can define on the set X. Recall that a relation on the set X is defined as any subset of $X \times X$. An equivalence relation is a relation which is reflexive, symmetric and transitive. Consider the relation R_x:

$x_1 R_x x_2 \leftrightarrow S(x_1) = S(x_2)$ for $x_1, x_2 \in X$

Clearly,

$x_1 R_x x_1$
$x_1 R_x x_2$ ==> $x_2 R_x x_1$
$x_1 R_x x_2$ and $x_2 R_x x_3$ ==> $x_1 R_x x_3$

Hence, R_x is an equivalence relation on the set X.

Now consider the relation R_y:

$x_1 R_y x_2$ <==> $x_1 \in S^{-1}(y)$ and $x_2 \in S^{-1}(y)$

where S^{-1} is the inverse set function associated with the function S. Again,

$x_1 R_y x_1$

$x_1 R_y x_2 \implies x_2 R_y x_1$

$x_1 R_y x_2$ and $x_2 R_y x3 \implies x_1 R_y x_3$

Hence, a program P producing an output y* on an input x* generates the following two equivalence classes:

$$X_x^* = \{x \in X: S(x) = S(x^*)\}$$

$$X_y^* = \{x \in X: S(x) = y^* = P(x^*)\}$$

That is, X_{x^*} is the set of all inputs for which a *correct* program produces the output S(x*) which is the correct output for the input x*. On the other hand, X_{y^*} is the set of all inputs for which a *correct* program produces the output $y^*=P(x^*)$ which may or may not be the correct output for the input x*. Now consider the following proposition:

Proposition: Given $x^* \in X$ and $y^* \in Y$ with $y^* = P(x^*)$, the program output y* is correct (more precisely, $P(x^*) = S(x^*)$) if and only if $X_{x^*} = X_{y^*}$.

Proof:

($\Rightarrow$) Let y* be correct. Then

$$y^* = P(x^*) = S(x^*)$$

Let $x \in X_{x^*}$. Then

$$S(x) = S(x^*) = y^* \Rightarrow x \in S^{-1}(y^*)$$

Hence,

$x \in X_{y^*}$

Now, let $x \in X_{y^*} \Rightarrow S(x) = y^* = P(x^*) = S(x^*)$

So $x \in X_{x^*}$

($\Leftarrow$) Let $X_{x^*} = X_{y^*}$

Now $x^* \in X_{x^*}$. Since $X_{x^*} = X_{y^*}$ this implies $x^* \in X_{y^*}$. Therefore,

$x^* \in S^{-1}(P(x^*))$

which implies

$S(x^*) = P(x^*)$

Q.E.D.

We have thus reduced the problem of checking the correctness of a program output on a given input to testing the equality of two sets. What we need is a characterization of X_{x^*} and X_{y^*} based on the specification S. If the number of elements in the sets X_{x^*} and X_{y^*} are infinite, then we need a random characterization. If these are available, then we can proceed with probabilistic interactive checkers.

Suppose that we have a random characterization of X_{x^*} and X_{y^*}. Given these random characterizations, we propose the following checker based *solely* on invoking the program P on random inputs from the subsets X_{x^*} and X_{y^*} and comparing the program outputs at these random inputs to the output being tested.

Generalized Checker. Given a program P supposedly satisfying a specification S, an input $x^* \in X$, and a corresponding output $y^* \in Y$, let

$S: X \rightarrow Y$ where $S(x) = y$ for $x \in X$, $y \in Y$

$P: X \rightarrow Y$ where $P(x) = y$ for $x \in X, y \in Y$

define

$X_x^* = \{x \in X: S(x) = S(x^*)\}$

$X_y^* = \{x \in X: S(x) = y^*\}$

where X_{x^*} is the set of all inputs on which a correct program would generate the same output as on x^*, and where X_{y^*} is the set of all inputs on which a correct program would have generated the output (possibly incorrect) y^*.

1. Choose randomly between X_{x^*} and X_{y^*} .
2. Choose a random element z from the set selected in step 1.
3. Invoke the program P on the input z selected in step 2.

   If $P(z) \neq y^*$, report INCORRECT

   If $P(z) = y^*$, go to step 1 up to k times.

If a program output is correct, then such a checker will always return correct. If the program output y^* is not correct, then the performance of the checker will be determined by the relationship between the element of $y^* \in Y$ and the following two subsets of Y:

$P(X_{x^*}) = \{y \in Y : y=P(x) \text{ for some } x \in X_{x^*}\}$

$P(X_{y^*}) = \{y \in Y : y=P(x) \text{ for some } x \in X_{y^*}\}$

If $P(X_{x^*}) \cup P(X_{y^*}) - \{y^*\} = \emptyset$, then such a checker will miss the incorrect output.

If $P(X_{x^*}) \cup P(X_{y^*}) - \{y^*\} \neq \emptyset$, then such a checker will eventually detect the incorrect output. The probability of the checker hitting an input providing inconsistency will depend on the relative cardinality of the two subsets of X

$\{x \in X_{x^*} \cup X_{y^*} : P(x)=y^*\}$

$\{x \in X_{x^*} \cup X_{y^*} : P(x) \neq y^*\}$

If the intersection of the subsets $P(X_{x^*})$ and $P(X_{y^*})$ is empty, then the efficiency of the checker is very high.

Example 2: Checker for a Sorter Program. Consider a program P which sorts a set of inputs according to a given relation, given the input x* and output y*:

$$x^* = \{x_1^*, x_2^*, \ldots, x_n^*\}$$
$$y^* = \{y_1^*, y_2^*, \ldots, y_n^*\}$$

The equivalence classes X_x^* and X_y^* are given by:

$$X_x^* = \{T(x^*): T(x^*) \text{ is a random permutation of } x^*\}$$

$$X_{y^*} = \varnothing, \quad \text{if } y^* \text{ is not ordered}$$
$$= \{T(y^*)\} \quad \text{otherwise}$$

where T is a random permutation of y*. Now the checker can be implemented with the algorithm:

1. Choose randomly by z* from x* or y*.
2. Select a random permutation T(z*).
3. If $P(T(z^*)) \neq y^*$, report INCORRECT
   $P(T(z^*)) = y^*$, repeat up to k times.

This checker will eventually detect all ordered incorrect answers for a sorter program. Unordered incorrect answers may also be detected depending on the relationship between the sets P (T(x*)) and P (T(y*)).

Example 3: Checker for a Least Squares Estimator Program. This example corresponds to the estimation module in the RSDIMU experiment. Consider a program P computing the least squares estimate

$$a = C^{\#} f$$

where f and a are the real-valued measurement and estimate vectors with dim (f) > dim (a), and $C^{\#}$ is the generalized inverse of the matrix C. Suppose, for a given measurement f*, the program P computes the estimate a*. In this case, the equivalence classes X_{f*} and X_{a*} are given by:

$$X_{f^*} = \left\{ f^* + \left[I - CC^{\#} \right] f : f \text{ is arbitrary} \right\}$$

$$X_{a^*} = \{ Ca^* + [I - CC\#]\, f: f \text{ is arbitrary}$$

In order to reduce the complexity of the checker, we employ the program P in the random characterization of these sets. For a correct program P, we have

$$X_{f^*} = \{ f^* + f - C\,P(f) : f \text{ arbitrary} \}$$

$$X_{a^*} = \{ Ca^* + f - C\,P(f) : f \text{ arbitrary} \}$$

Now the checker can be implemented with the algorithm

1. Choose randomly either X_{f*} or X_{a*}
2. Choose a random element f from the set selected in step 1
3. If $P(f) \neq a^*$, report INCORRECT
   $P(f) = a^*$, repeat up to k times.

Note that this checker employs the program in the selection of a random element from two equivalence classes.

Example 4: Checker for an Eigenvalue Computation. Consider a program P which computes the eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$ for real valued square matrices A:

$$S: R^n \times R^n \rightarrow C^n \text{ with } S(A) = \{\lambda_1, ..., \lambda_n\}$$

Suppose for a given matrix A^*, the program P computes the eigenvalues $\{\lambda_1^*, ..., \lambda_n^*\}$. The equivalence classes are given by:

$$X_{A^*} = \{PA^* P^{-1}: \text{ P is arbitrary, nonsingular}\}$$

$$X_{\lambda^*} = \{PJ^* P^{-1}: \text{ P is arbitrary, nonsingular}\}$$

with J^* = block diagonal $\{J_1^*, ..., J_p^*\}$ where J_i^* are the Jordan blocks associated with the eigenvalues $\lambda_1^*, ..., \lambda_n^*$. That is, the equivalence classes consist of random similarity transformations of the input matrix and the Jordan normal form of the eigenvalue outputs.

4. RELIABILITY ANALYSIS

In this section we present the results of our reliability analysis using the data obtained from the RSDIMU experiment.

4.1. N-Version vs. Recovery Block Structures

In this section we present a comparison of the reliability of N-Version structures and Recovery Block Structures. We use models of the reliability of structures built from an infinite population of versions characterized by a failure intensity distribution g. We then compare the performance of these two types of structures using the failure intensity distribution obtained from the RSDIMU experiment.

4.1.1. N-Version Structure Reliability

In an N-Version fault tolerant structure, N independently designed similar software components (where N is an odd integer) are executed and the results are passed to a voter. The majority consensus output is the output of the fault tolerant structure. The probability of failure of such a structure p_k is shown by [Eckhardt and Lee, 1985] to be:

$$p_k = \sum_{i=0}^{N} \left(\sum_{j=m}^{k} \binom{k}{j} \left(\frac{i}{N}\right)^j \left(1 - \frac{i}{N}\right)^{k-j} \right) g\left(\frac{i}{N}\right), \text{ where } m = \frac{(k+1)}{2} \qquad 4.1.1$$

4.1.2. Recovery Block Structure Reliability

A fault tolerant recovery block structure consists of k versions of redundant software (one primary alternate and k-1 supplementary alternates) to be executed serially with sequential acceptance checks performed between version executions.

Clearly, the overall reliability of a recovery block structure not only depends on the reliability of the individual component versions, but also on the reliability of the acceptance check routine. Our definition of acceptance check reliability encompasses not only the reliability of the

acceptance check software, but also the coverage of the acceptance check algorithm in distinguishing between correct and incorrect version outputs.

4.1.2.1. Recovery Block Structure Reliability with Perfect Acceptance Check

We begin our analysis of recovery block structures in this section by assuming the existence of a perfect acceptance check routine. That is, an acceptance check routine containing no software errors with the additional property that for a given input-output set of data:

1. P[acceptance check declares output incorrect | output is correct] = 0
2. P[acceptance check declares output correct | output is incorrect] = 0

That is, the first probability refers to the false alarm rate while the latter refers to the probability of missed detection of the acceptance check test. Later sections will approach the problem assuming non-zero failure probability.

Assume that we have a version population which is characterized by the failure intensity distribution:

$$g\left(\frac{i}{N}\right),\ 1 \le i \le N \qquad 4.1.2$$

Under this assumption, the probability of failure of a k-version recovery block structure with perfect acceptance check is:

$$p_k = \sum_{i=0}^{N} \left(\frac{i}{N}\right)^k g\left(\frac{i}{N}\right) \qquad 4.1.3$$

4.1.2.2. Recovery Block Structure Reliability with Imperfect Acceptance Check

In order to model the reliability of a recovery block structure with an imperfect acceptance check, we make the following assumptions:

1. P[acceptance check declares output incorrect | output is correct] = 0
2. Acceptance check failures are independent of component version failures.

The first assumption concerning the false alarm rate is a reasonable one for most acceptance checks. The second assumption's validity is a function of the specific application under consideration. Under the second assumption:

P[acceptance check declares output correct | output is incorrect]
= P[acceptance check fails]
= ε

where ε denotes the probability of missed detection of the acceptance check. Under these assumptions we are able to determine the probability q_k that a k-version recovery block structure will fail. If we assume that k-versions are chosen from a population which is characterized by the failure intensity distribution of 4.1.2, then the probability of failure is

$$q_k(\varepsilon) = p_k (1 - \varepsilon)^{k-1} + \varepsilon \sum_{i=1}^{k-1} p_i (1 - \varepsilon)^{i-1} \qquad 4.1.4$$

where p_i is given by 4.1.3.

4.1.3. Comparison of Structures for RSDIMU Experiment Version Population

In this section we compare the reliability of N-Version and Recovery Block fault tolerant structures. In particular we will consider a population of programs which is characterized by the failure intensity distribution obtained from the RSDIMU experiment. We have combined the individual cases $S_{0,0}$ through $S_{1,2}$ to obtain the failure intensity distribution of table 4.1.3.1. Using equations 4.1.1 and 4.1.4 we are able to compare the predicted effectiveness of N-Version and recovery block structures built from a realistic population.

θ	$g(\theta)$
0/20	9.1802e-1
1/20	5.8575e-2
2/20	2.2995e-2
3/20	8.3628e-5
4/20	1.5748e-4
5/20	8.7972e-5
6/20	4.5615e-5
7/20	2.2808e-5
8/20	6.5164e-6
9/20-20/20	0

Table 4.1.3.1: Failure Intensity Distribution Obtained from RSDIMU Experiment.

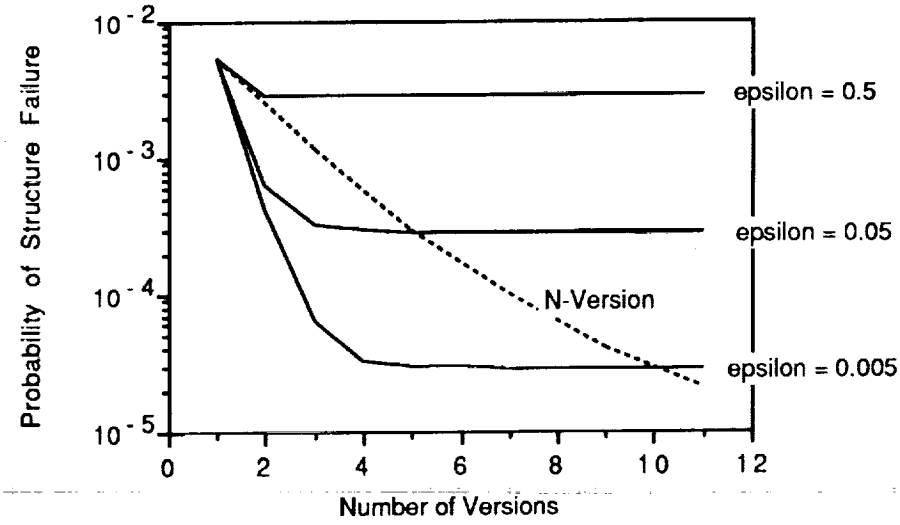

Figure 4.1.1: Comparison of N-Version and Recovery Block Structure Reliability.

The result of modelling an N-Version structure with a recovery block structure with an imperfect acceptance check is shown in figure 4.1.1. Here we have sampled the reliability curve

for a recovery block structure using acceptance check failure probabilities -- ε -- of 50%, 5%, and 0.5%. It is immediately apparent that a recovery block structure whose acceptance check is no more than 50% reliable is no more effective than an N-Version structure no matter how many versions are used.

With more reliable acceptance checks, however, there is an advantage to choosing to build a recovery block structure over an N-Version structure when a relatively small number of versions are involved. With the population of table 4.1.3.1 the decision point of which fault tolerant structure to use is:

	use N-Version Structure	use Recovery Block Structure
for $\varepsilon = 5\%$:	$k > 5$	$k < 5$
for $\varepsilon = 0.5\%$:	$k > 10$	$k < 10$

If three versions are to be used in building a structure, the failure probability of our acceptance check would have to be less than approximately 0.195 in order to warrant the use of N-Version programming instead of recovery blocks.

This leads us to believe that if it is possible to develop a reasonably reliable acceptance check whose failure behavior is independent of the programs that it is checking, then the decision of whether to use N-Version programming or recovery blocks depends upon the number of versions that are to be employed. In the case where it is possible to develop a large number of versions to include in the structure, then N-Version programming should be employed. On the other hand, if only a small number of versions can be developed then it is better to use recovery blocks.

The development of a recovery block model for which the acceptance check is not assumed to be independent would be very advantageous. This would make it possible to decide which fault tolerant methodology to use in the case where an independent acceptance check is not possible.

4.2. Reliability of Software Fault Tolerant Structures Under Diverse Methodologies

Here we examine the reliability of fault tolerant software structures in order to determine how best to exploit the independence which exists between subpopulations of the twenty RSDIMU versions. This will help us to determine the best ways to exploit independence resulting from diverse development methodologies.

The RSDIMU programs were developed by independent development teams at four development sites. They were then certified at three different certification sites. The different sites lead to the determination of two different subpopulation classifications. These are classification according to development site and classification according to certification site. In addition, in examining the individual reliabilities of the programs we have identified three clearly defined reliability subpopulations. We refer to these subpopulations as high, medium, and low reliability groups. By separating the programs into these three groups independent of development and certification site, we are able to fabricate diverse subpopulations.

In the following sections we model the reliability of fault tolerant software structures which are built using the following techniques:

- *subpopulation structures*: Structures built using only programs from within each specific subpopulation.
- *homogenous structures:* Structures built using the aggregate of all subpopulations.

In comparing the subpopulation structures with the homogenous structure we must be careful only to generalize the end results. This comparison is not meaningful when we are searching for the sources of the software failures. In other words we cannot say that the homogenous structure showed a relatively poor reliability because of subtle dependencies between the methodologies at different development sites or certification sites. This is invalid because cross correlation between programs at different development sites could be due to planned factors in the experiment, such as the use of a common specification and the fact that versions from different development sites were certified at the same certification site.

4.2.1. Diversity of Development Site

The failure intensity distribution of Table 4.2.1.1. results from separating the versions into four groups according to their development site.

failure intensity	development site				
	ncsu	ucla	uiuc	uva	homogenous
0	920616	866187	878637	920379	845268
1	130	53982	42025	262	53933
2	0	566	84	105	21173
3	0	11	0	0	77
4	0	0	0	0	145
5	0	0	0	0	81
6					42
7					21
8					6
9-20					0

Table 4.2.1.1: Failure Intensity Distribution by Development Site.

The average reliability varies widely, and sites *ucla* and *uiuc* have a very high number of single version failures. Of the single version failures in the *ucla* subpopulation, 99.1% are due to the failure of version *uclab*. Of the single version failures in the *uiuc* subpopulation, 99.9% are due to the failure of version *uiuca*.

Figure 4.2.1.1 shows the improvement of reliability of structures formed from subpopulations and the homogenous population. The uva subpopulation structure showed less than an order of magnitude improvement in reliability even though its failure intensity distribution in table 4.2.1.1 shows fewer high-intensity errors than the ucla subpopulation. The reason for this is that the ucla subpopulation has a very large number of failures of intensity 1. Note that the reliability improvement of the ncsu, ucla, and uiuc structures are nearly identical even though the ncsu failure intensity distribution shows significantly fewer coincident failures and higher average reliability.

A significant feature of figure 4.2.1.1 is the remarkable improvement of the homogenous structure compared to that of the subpopulation structures. This suggests that the dependencies

between versions in the subpopulations are not as significant when those subpopulations are combined. This is because there is more dependent failure behavior between versions from the same subpopulation that from different subpopulations.

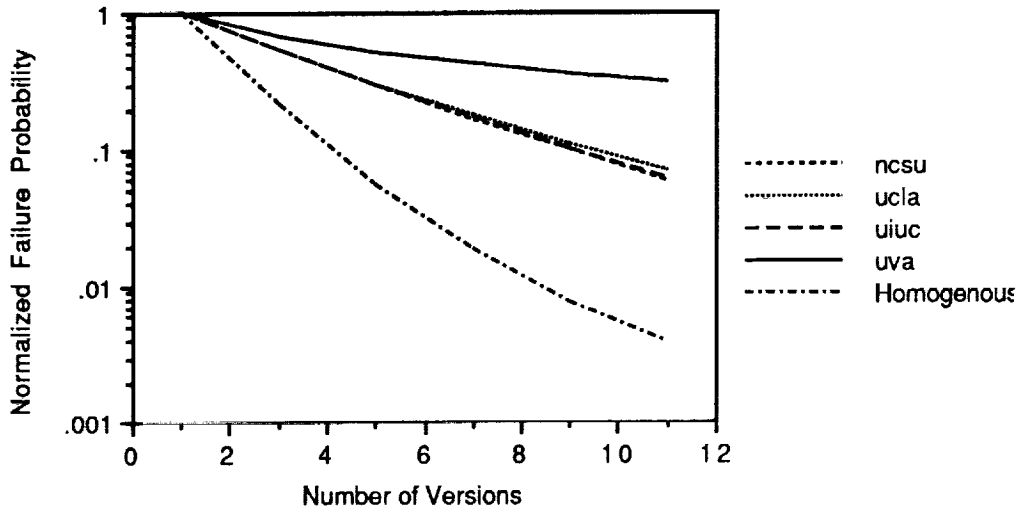

Figure 4.2.1.1: Coincident Error Model of Reliability
of N-Version Structures Built from Development Subpopulations.

Figure 4.2.1.2. shows the *finite population model* of structures built from development subpopulations. Each subpopulation structure in figure 4.2.1.2. performed as would be expected from the failure intensity distributions of table 4.2.1.1. In this figure the lines which drop off the plot (ncsu at $N \cong 3$ and uiuc and uva at $N \cong 5$) are considered to have reliability 1.0 at this point. This is because no coincident failures were observed that could cause failure of any structure with the N versions. The homogenous structure shows a smaller improvement than the subpopulations in the finite population mode. By considering the populations to be composed only of the versions developed in the RSDIMU experiment, we exaggerate the effects of the low reliability versions on the resulting structure. In the infinite population of the coincident error model, the effect of this small number of low reliability versions is overcome by the independence which exists in the majority of the versions.

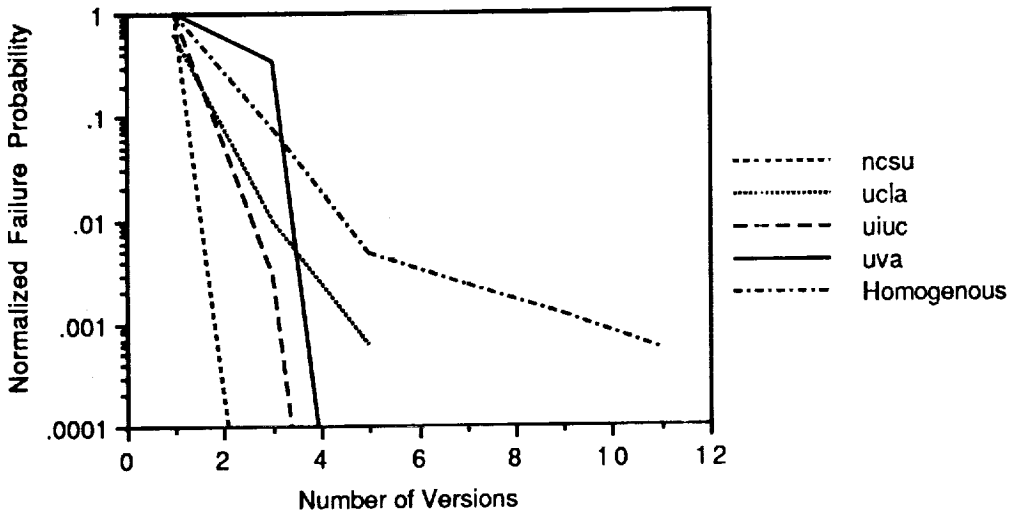

Figure 4.2.1.2: Finite Population Model of Reliability
of N-Version Structures Built from Development Subpopulations.

4.2.2. **Diversity of Certification Site**

In order to determine the effects of certification site diversity on the independence of failures we divided the versions into three categories according to table 4.2.2.1.

ncsu	ucsb	uva
ncsuc	uclac	ncsua
ncsud	uclae	ncsub
ncsue	uiuca	uclaa
uclad	uiucb	uclab
uiucc	uiucd	uiuce
uvaa	uvab	uvad
uvac		uvae

Table 4.2.2.1. Certification Sites for RSDIMU Programs.

The failure intensity distribution of table 4.2.2.2. results from separating these versions into subpopulations by certification site.

failure intensity	certification site			
	ncsu	uva	ucsb	homogenous
0	920102	867192	877872	845268
1	330	53554	42545	53933
2	313	0	224	21173
3	1	0	14	77
4	0	0	56	145
5	0	0	34	81
6	0	0	0	42
7	0		0	21
8				6
9-20				0

Table 4.2.2.2. Failure Intensity Distribution by Certification Site.

Here we see that the highest intensity failures are all within the same certification site subpopulation. The average reliability of the subpopulations uva and ucsb are similar while the ncsu subpopulation exhibits a much higher average reliability. 99.9% of the uva failures were caused by version uclab and 98.8% of the single version failures in ucsb were caused by version uiuca.

In figure 4.2.2.1 we see some of the same behavior that we saw in figure 4.2.1.1. The ncsu subpopulation structure shows little improvement even though there are relatively few coincident errors. We also see that for $N \leq 8$ the uva and ucsb subpopulations exhibit less than an order of magnitude reliability improvement even though the failure intensity distributions for these subpopulations show dramatic improvement for failure intensities greater than 1. This is most dramatic in the uva subpopulation which shows better than independent behavior. Once again, the homogenous structure shows more improvement than the subpopulation structures.

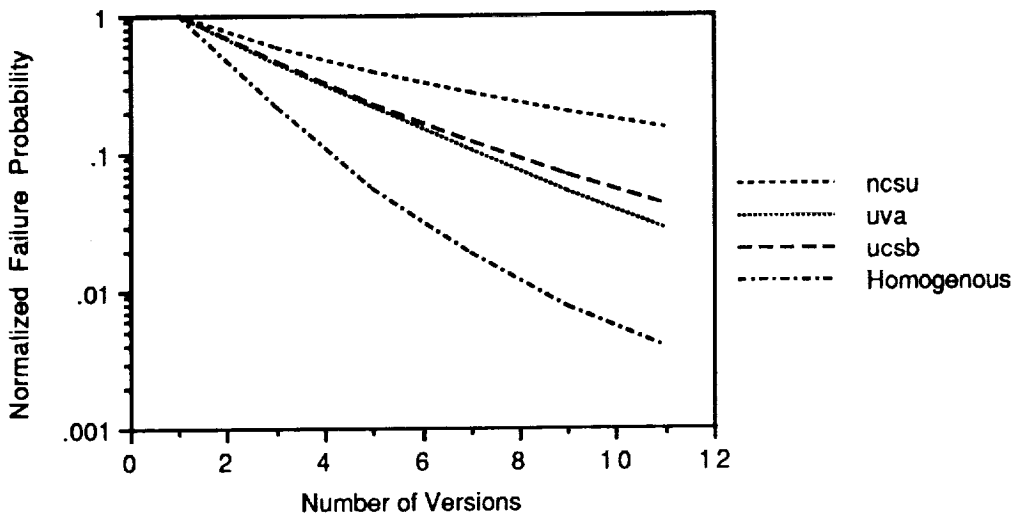

Figure 4.2.2.1: Coincident Model of Reliability
of N-Version Structures Built from Certification Subpopulations.

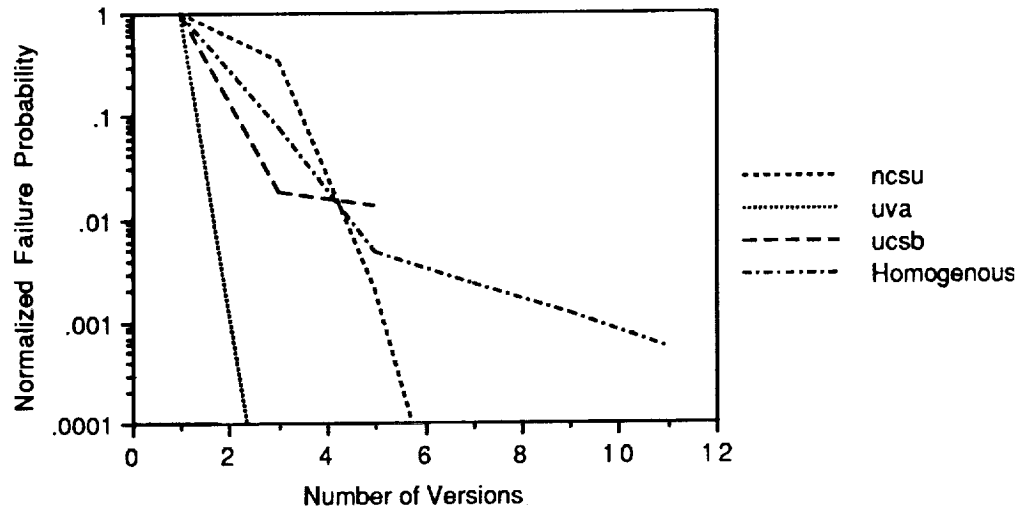

Figure 4.2.2.2: Finite Population Model of Reliability
of N-Version Structures Built from Certification Subpopulations.

As in figure 4.2.2.1, figure 4.2.2.2 shows that the homogenous structure performs poorly in relation to the subpopulation structures for the same reasons as have been discussed.

4.2.3. Diversity of Version Reliability

In order to determine the effects of version reliability diversity upon diverse structure reliability we created an artificial grouping of the versions according to their reliability. The version reliabilities of the 20 RSDIMU programs are plotted in figure 4.2.3.1.

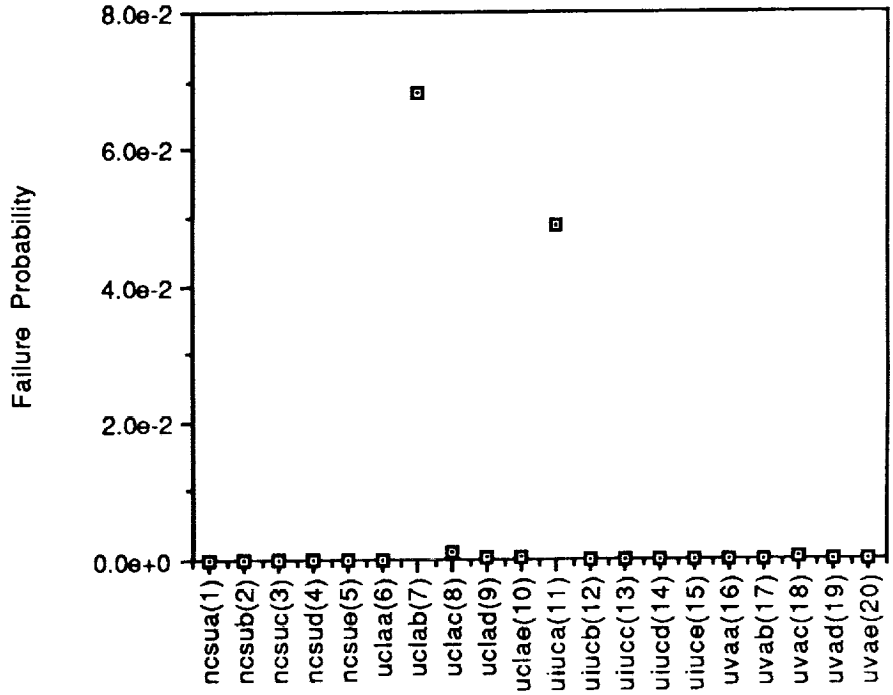

Figure 4.2.3.1: Scatter Plot of RSDIMU Version Reliability.

Because of the extremely low reliability of versions uclab and uiuca it is difficult to determine more than two groups. A better view of the intermediate reliability subpopulation is shown in figure 4.2.3.2. In this figure versions uclab and uiuca have been removed and the Y-axis has been scaled accordingly.

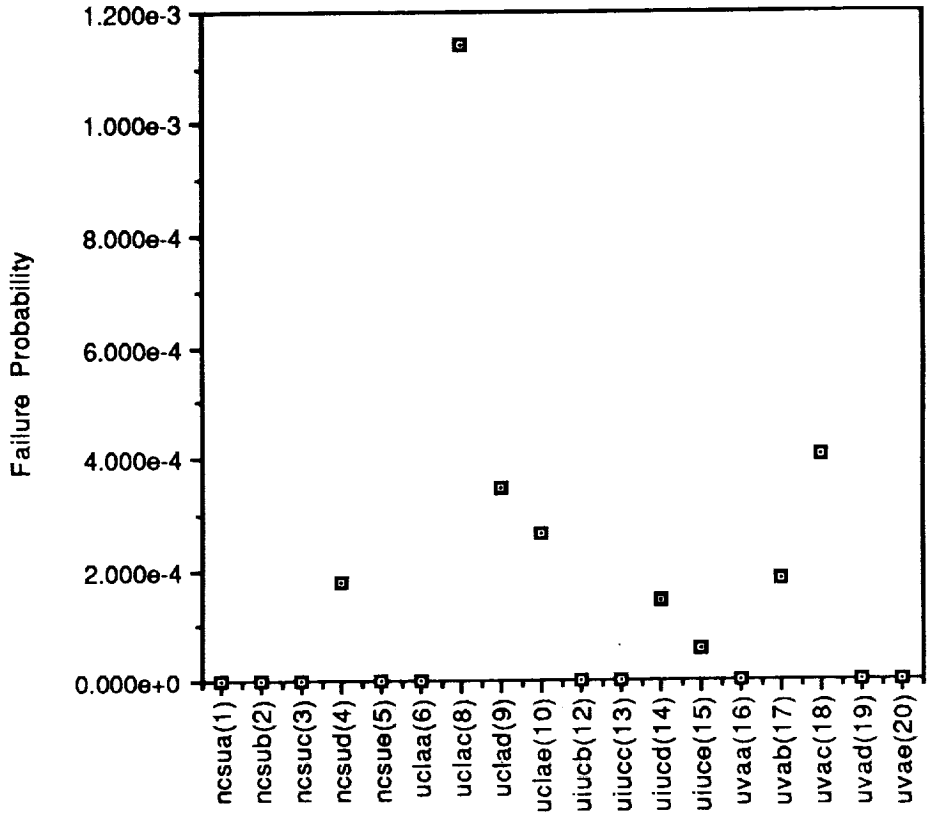

Figure 4.2.3.2: Scatter Plot of RSDIMU Version Reliability with versions uclab and uiuca removed.

With the information presented in these two figures we divided the versions into three subpopulations based upon the individual version reliabilities. These subpopulations are shown in table 4.2.3.1. The variation of reliabilities and the number of versions differ between groups, however the average reliabilities of these subpopulations is roughly log-linear as shown in figure 4.2.3.3.

High	Medium	Low
ncsua	ncsud	uclab
ncsub	uclad	uiuca
ncsuc	uclac	uclac
ncsue	uiucd	
uclaa	uiucc	
uiucb	uvab	
uiucc	uvac	
uvaa		
avad		
uvae		

Table 4.2.3.1: Version Subpopulations Based upon version reliability.

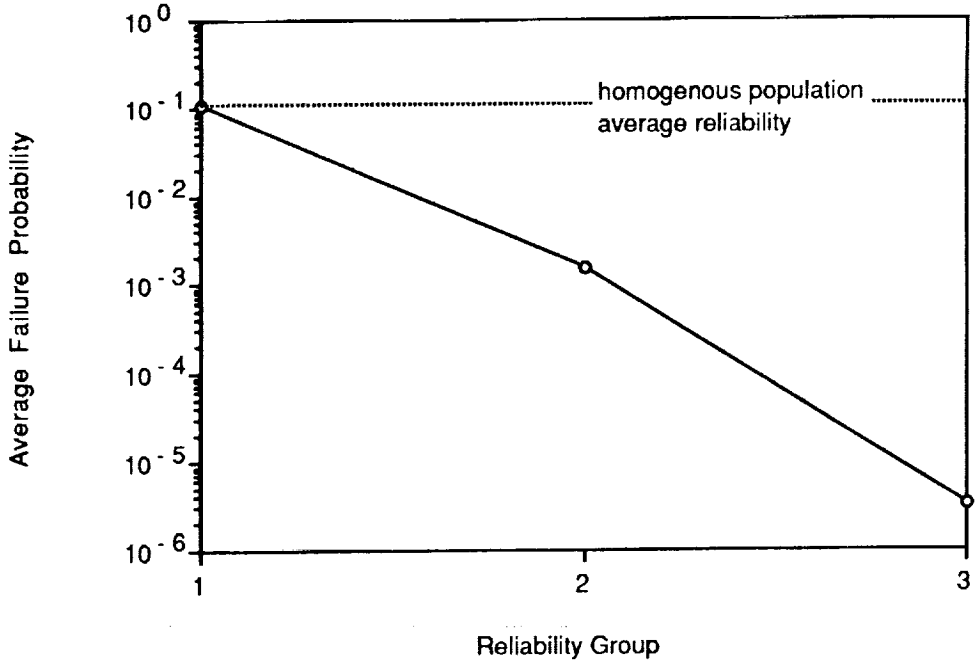

Figure 4.2.3.3: Average Failure Probability of Reliability Group Subpopulations.

failure intensity	Reliability Group			
	High	Medium	Low	homogenous
0	920744	920020	845358	845268
1	1	358	54243	53933
2	1	215	21135	21173
3	0	57	10	77
4	0	43		145
5	0	53		81
6	0	0		42
7	0	0		21
8	0			6
9-20	0			0

Table 4.2.3.2: Failure Intensity Distributions by Reliability Groups

Table 4.2.3.2: Shows the failure intensity distributions resulting from partitioning the versions into reliability subpopulations. It is difficult to draw any conclusions based upon the failure intensity distributions as there are a different number of versions involved in each subpopulation. The low reliability subpopulation exhibits 10 failures involving all versions in the subpopulation while the medium reliability subpopulation has none. The high reliability group has three failures and two of these were correlated. The high reliability group also contains 7 versions with reliability 1.0 (no failures reported).

Figure 4.2.3.4: Shows the result of the *coincident error model* using these failure intensity distributions. The reliability improvement is very small for the low and medium reliability subpopulations. In the case of the low reliability subpopulation this is due to the high proportion of correlated failures among the versions in the group. Since there were reported failures involving all versions, this carries through to the characterization of the infinite population, yielding no improvement. The improvement in the medium reliability subpopulation is suppressed for basically the same reason. Only in the highest reliability group do we achieve any improvement. The single order of magnitude improvement for an 11 version structure is quite small considering the expense involved in developing this number programs of such high version reliability.

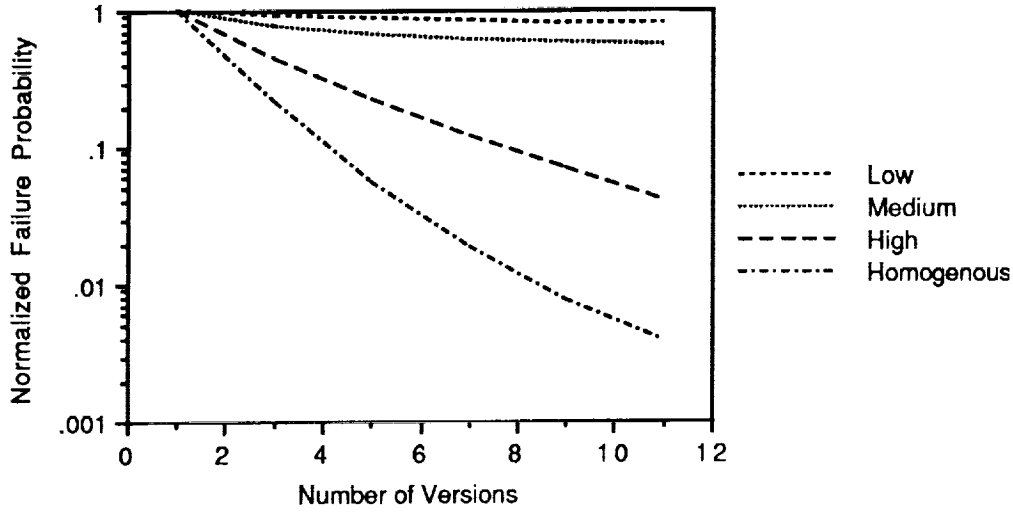

Figure 4.2.3.4: Coincident Model of Reliability
of N-Version Structures Built from Reliability Subpopulations.

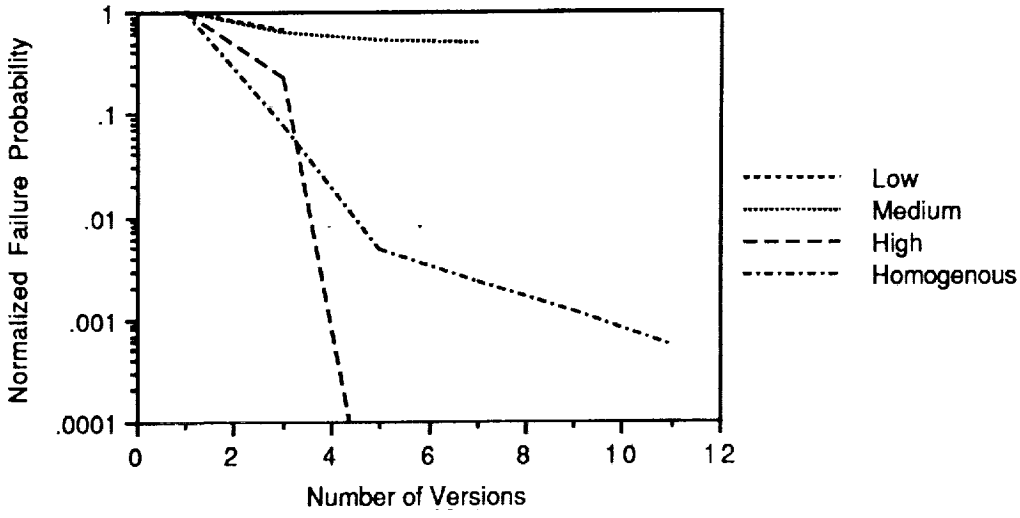

Figure 4.2.3.5: Finite Population Model of Reliability
of N-Version Structures Built from Reliability Subpopulations.

Once again, the homogenous structure shows a better improvement than the subpopulation structures. Here, however, it is because we have forced dependencies in the subpopulations by characterizing them by their average reliability.

The *finite population model* of figure 4.2.3.5 shows similar results to that of figure 4.2.3.4. Due to the varying size of the subpopulations, the prediction of structure reliability is not valid for larger values of N in some subpopulations. This occurs for small N in the low reliability group since there are only three versions in this subpopulation.

4.2.4. Results of Subpopulation Diversity Reliability Analysis

The analysis presented in the preceding sections contains some surprising observations. In addition there are some conclusions that lead to further study of diversity and N-Version reliability modelling. One significant observation is the improvement of homogenous structures over that of subpopulation structures. In all examples of subpopulations the *coincident error model* predicts significant improvement of the homogenous structure in the range of $1 \leq N \leq 11$ versions. While the subpopulation structures usually show some improvement in this range, this improvement is limited to approximately 1 order of magnitude while the homogenous structure improves by approximately 2 orders of magnitude. This leads us to believe that a population which incorporates versions which were produced by different methodologies will benefit more from N-Version programming than a population whose versions were produced by similar methodologies.

The finite population model of the homogenous structure exhibits smaller reliability improvement in the case of the homogenous population than any of the subpopulation structures. This result contradicts the predictions of the infinite population model. By limiting the population of versions, the finite population model exaggerates the effects of the extremely low reliability versions in the RSDIMU population.

4.3. Diverse N-Version Fault Tolerant Structures

Here we present the idea of a diverse N-Version fault tolerant software structure. Such a structure is intended to take advantage of independence which results from using different development methodologies to develop versions which are combined into a single fault tolerant software structure. We use the RSDIMU software versions to examine the reliability of these

diverse structures and to predict the possible advantages of diverse structures over homogenous structures.

4.3.1. Definition and Modelling of Diverse Structures

A diverse software structure is composed of N different programs developed under M different methodologies. The more diverse the methodologies are, the greater the advantage of using diverse structures will be. A diverse structure has between $\lfloor \frac{N}{M} \rfloor$ and $\lceil \frac{N}{M} \rceil$ versions from each of the M methodologies. In the case when N = kM where k is an integer, the method of choosing versions for a diverse structure is uniquely defined. However, where k is not an integer the method of choosing versions for a diverse structure is not uniquely defined. In such a case there will be N $\boxed{\text{mod}}$ M different methods of choosing versions. These methods represent different numbers of versions from each methodology to incorporate into the diverse structure.

In [Littlewood and Miller 1990] the authors present an infinite population model for diverse structures where $N \leq M$. Consider the set Π_A of programs developed under methodology A. Similarly consider the sets Π_B and Π_C. For N=M=3 the probability of failure of a diverse structure P_3 is defined by

$$P_3 = E(\theta_A\theta_B) + E(\theta_A\theta_C) + E(\theta_B\theta_C) - 3E(\theta_A\theta_B\theta_C) \qquad 4.3.1$$

This represents a voting structure which fails whenever a majority of its versions fail. The expected values of 4.3.1. are defined as follows:

$$E(\theta_A\theta_B) \qquad 4.3.2$$

$$= \sum_{i=1}^{k} \sum_{\pi_A \in \Pi_A} \sum_{\pi_B \in \Pi_B} v(\pi_A, x_i)\, v(\pi_B, x_i)\, S(\pi_A)\, S(\pi_B)$$

$$E(\theta_A\theta_B\theta_C) \qquad 4.3.3$$

$$= \sum_{i=1}^{k} \sum_{\pi_A \in \Pi_A} \sum_{\pi_B \in \Pi_B} \sum_{\pi_C \in \Pi_C} v(\pi_A, x_i)\, v(\pi_B, x_i)\, v(\pi_C, x_i)\, S(\pi_A)\, S(\pi_B)\, S(\pi_C)$$

Where $S(\pi_A)$ is the probability of developing a program π_A under methodology A and $v(\pi_A, X_i)$ is equal to 1 if program π_A fails on input X_i.

For the infinite population, we can obtain these expected values using a *joint failure intensity distribution.* The joint failure intensity distribution $\hat{g}$ is defined as:

$$\hat{g}(x,y,z) = \left\langle \textit{total \# times that} \left\{ \begin{array}{l} \textit{x versions from A} \\ \textit{y versions from B} \\ \textit{z versions from C} \end{array} \right\} \textit{fail over all inputs} \right\rangle \qquad 4.3.4$$

Using this definition, equations 4.3.2. and 4.3.3. become:

$$E(\theta_A\theta_B) = \sum_{i=1}^{|A|} \sum_{j=1}^{|B|} \sum_{k=1}^{|C|} \hat{g}(i,j,k)\left(\frac{i}{|A|}\right)\left(\frac{j}{|B|}\right) \qquad 4.3.5$$

$$E(\theta_A\theta_B\theta_C) = \sum_{i=1}^{|A|} \sum_{j=1}^{|B|} \sum_{k=1}^{|C|} \hat{g}(i,j,k)\left(\frac{i}{|A|}\right)\left(\frac{j}{|B|}\right)\left(\frac{k}{|C|}\right) \qquad 4.3.6$$

Using 4.3.1, 4.3.5, and 4.3.6 it is possible to predict the performance of a diverse 3 version fault tolerant software structure built from infinite subpopulations A, B, and C which are characterized by $\hat{g}$.

If we consider only the 20 versions of the RSDIMU experiment, we can predict the reliability of a randomly chosen diverse structure. We first partition the set of 20 programs into

disjoint subsets. Consider A, B, and C as disjoint subsets of the 20 programs. Let Ψ be the set of all diverse structures which may be chosen from the 3 sets A, B, and C. Let S(J) be a function which characterizes a set of programs J which form a structure by giving a vector (x, y, z) where:

$$\begin{aligned} S_x(J) &= |J \cap A| \\ S_y(J) &= |J \cap B| \\ S_z(J) &= |J \cap C| \end{aligned} \qquad 4.3.7$$

Using (4.3.7) we can partition Ψ into disjoint subsets Ψ_i such that:

$$\Psi_i = \left\{ J \mid J_1, J_2 \in \Psi \wedge S(J_1) = S(J_2) \right\}, \text{ for } 0 \le i \le N \boxed{\text{mod}} 3 \qquad 4.3.8$$

If we assume no preference between sets A, B, and C in producing a diverse structure we can estimate the reliability of a 3-Version structure $\widetilde{P}_k$ in terms of Ψ_i as

$$\widetilde{P}_N = \sum_{i=0}^{N \boxed{\text{mod}} 3} \frac{1}{(N \boxed{\text{mod}} 3) + 1} \widetilde{P}'_N(\Psi_i) \qquad 4.3.9$$

...where $\widetilde{P}'_N(\Psi_i)$ is an estimate of the reliability of a randomly chosen diverse structure $J \in \Psi_i$ composed of a unique number of versions from each of the sets A, B, and C.

In order to define $\widetilde{P}'_N$ we refer to the estimator of [Eckhardt, et. al. 1990]. Let

$$u_j(x,l) = \begin{cases} 1 & \text{if l versions in set J fail on input x} \\ 0 & \text{if otherwise.} \end{cases} \qquad 4.3.10$$

Given a diverse structure j we know that the proportion of inputs on which a structure j fails is

$$\frac{1}{k} \sum_{i=1}^{k} \sum_{l=m}^{N} u_j(x_i, l) \quad \text{where } m = \frac{n+1}{2} \qquad 4.3.11$$

Using 4.3.11 we can define $\tilde{P}'_N$ in terms of u_j as:

$$\tilde{P}'_N(\Psi) = \frac{\sum_{j \in \Psi} \sum_{i=1}^{k} \sum_{l=m}^{N} u_j(x_i, l)}{k|\Psi|} \qquad 4.3.12$$

In 4.3.12 we know $S(j_i) = S(j_j)$ for all $j_i, j_j \in \Psi$ so it is a straightforward task to produce the set Ψ. For the version subpopulations of section 4.2. the size of Ψ is small enough to fit in computer memory so that it is possible to directly compute the numerator of 4.3.12 in a reasonable amount of time. Thus we can compute $\tilde{P}_N$ of 4.3.9 by partitioning Ψ into Ψ_i and directly computing $\tilde{P}'_N(\Psi_i)$.

The graph of figure 4.3.1 shows the results of our analysis using the Littlewood-Miller model for a 3-Version diverse structure. If we assume that all of the versions in a homogenous population fail independently we can obtain a prediction of the reliability of an N-Version structure using only the reliabilities of the 20 RSDIMU versions. The resulting reliability is labeled *independent* in figure 4.3.1. This represents the upper bound of reliability for any type of N-Version structure. The lower bound of reliability for a diverse N-Version structure was shown by Littlewood and Miller to be the Eckhardt-Lee model of the homogenous population, since Eckhardt and Lee assumes

$$\mathrm{Cov}(\theta_A, \theta_B) = \mathrm{Var}(\theta_A) = \mathrm{Var}(\theta_B)$$

This lower bound is labeled *coincident* also in figure 4.3.1. Note that if we desire to choose multiple versions from the same population, the Littlewood-Miller model suffers from the same identical distribution assumption as the Eckhardt-Lee model. [Eckhardt and Lee, 1985]

In order to investigate the effects of diverse methodologies we consider the site diversity discussed in section 4.2, specifically development site diversity and certification site diversity.

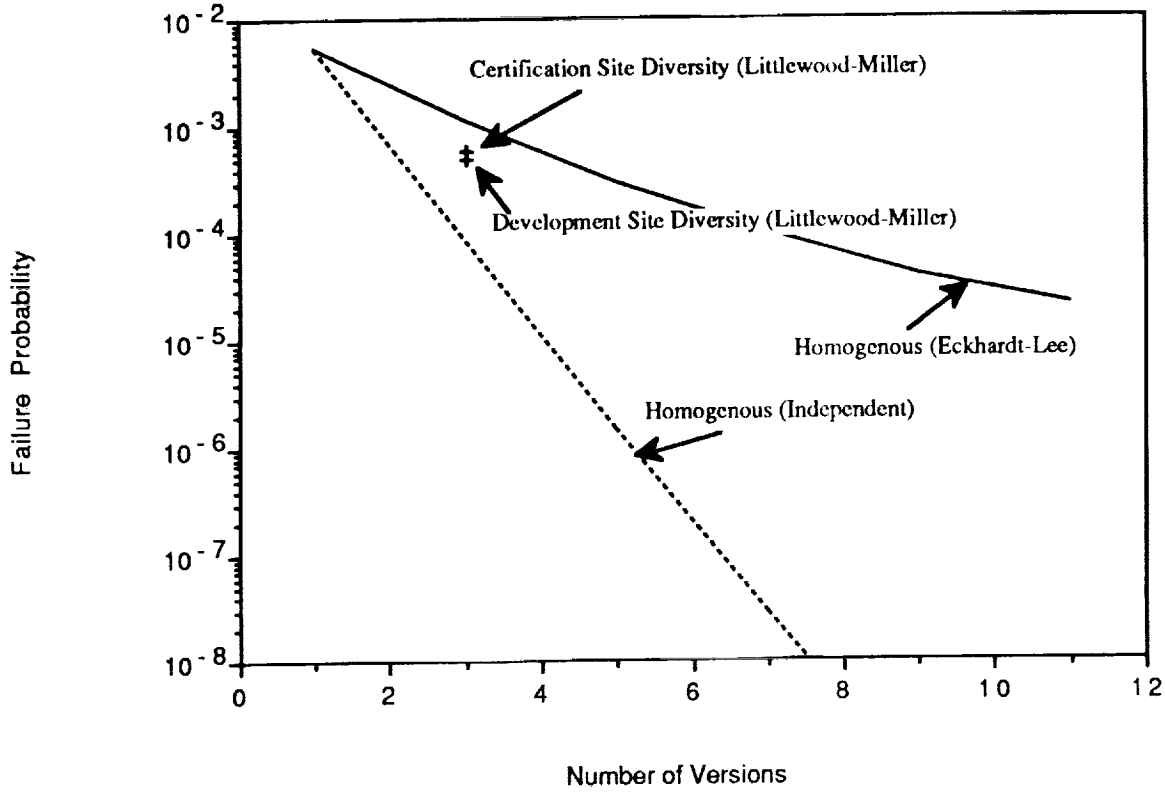

Figure 4.3.1: Comparison of Infinite Population Models for Diverse and Homogenous Structures.

In figure 4.3.1 we see that the reliability of diverse structures lies within the upper and lower bounds provided by the coincident and independent error models.This modest reliability improvement is expected since the different sites do not necessarily represent diverse methodologies and indeed certain common methodologies were forced upon the sites. In addition we have compared the finite population predictors to give an unbiased estimate of structure reliability based solely upon the population of 20 RSDIMU versions.

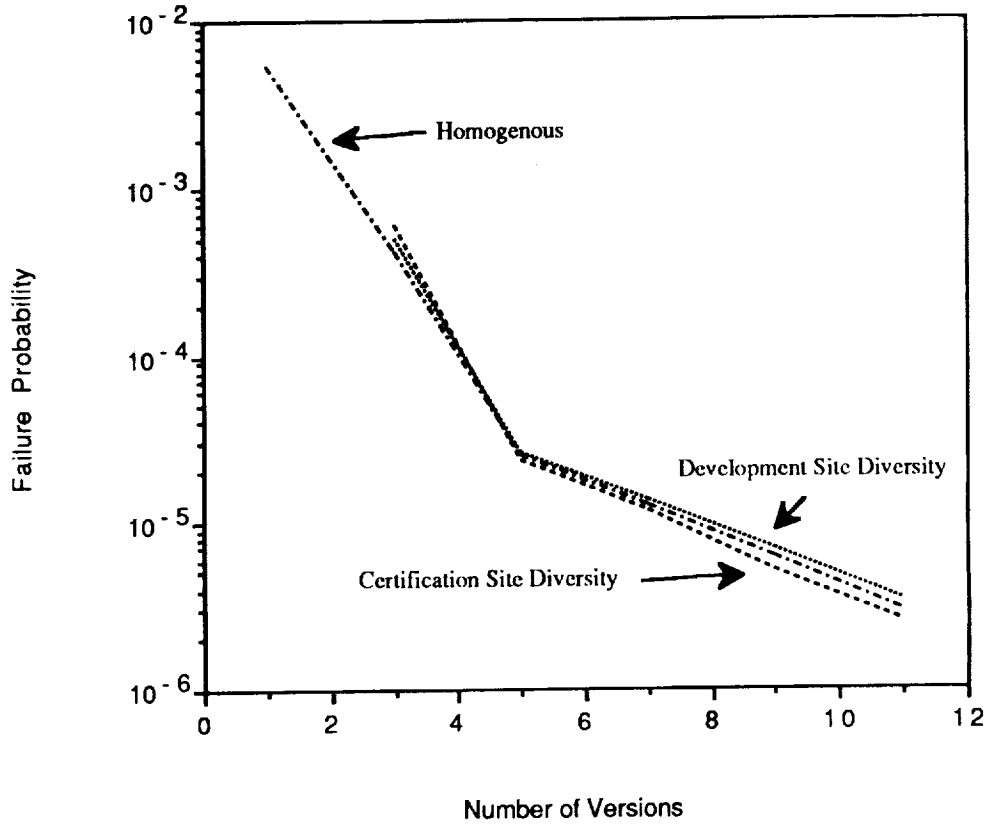

Figure 4.3.2: Comparison of Finite Population Structure Reliability for Diverse and Homogenous Structures.

Figure 4.3.2. shows a comparison of the certification site and development site diverse structures and the homogenous structure. The result is that the reliability of the homogenous structure is virtually identical to that of the certification and development structures.

Table 4.3.1 shows the effect of the selection criterion upon the certification site structure. The inclusion of both high and low reliability structures in the homogenous structure make its reliability medium while the inclusion of only the medium structures in the certification site diverse structure yields a medium reliability prediction.

The result is that the certification site structure arrives at the same reliability prediction as the homogenous structure via an entirely different selection criterion.

Number of Versions to Form Structure			Predicted Structure Reliability				
ncsu	ucsb	uva	high	med	low	certification site structure	homogenous structure
3	0	0	•				•
0	3	0			•		•
0	0	3					•
2	1	0	•				•
2	0	1	•				•
0	2	1			•		•
1	2	0			•		•
0	1	2			•		•
1	0	2			•		•
1	1	1		•		•	•

Table 4.3.1: Version Selection for Certification Site Diverse Structure Compared to Homogenous Structure.

5. CONCLUSIONS

In examining the types of errors made by programmers, we have discovered several different relationships which hold between these errors. These relationships indicate the eventual failure behavior within a group of programs. *Conceptually-Related* errors can cause the programs that contain them to fail dependently, however this is not always the case. We have shown cases where conceptually related errors have input domains which are disjoint or have very little overlap. We have also shown that some *conceptually-unrelated* errors also cause programs containing them to fail dependently. This is the case when the input domains for errors overlap significantly.

Diverse development methodologies may minimize the occurrence of conceptually-related errors. Diverse testing methodologies may reduce the occurrence of conceptually-unrelated errors which cause coincedent failures. It is not enough to relate errors conceptually. Programmer diversity may cause different levels of failure correlation for programs containing the same conceptual error, thus indicating the usefulness of the independent development paradigms (used in the RSDIMU experiment) in reducing identical and wrong errors.

We have described the notion of a *diverse N-Version structure* in which the redundant components are produced by diverse methodologies. We have described two models for predicting the reliability of diverse N-Version structures and used these models in conjunction with the data obtained from the RSDIMU experiment. Infinite population models indicate that diverse methodologies might be developed which will offer an improvement of an order of magnitude. Finite population models indicate that diverse structures perform the same as a structure which ignores such diversity. The discrepancy between the reliability predictions given by these models can only be resolved through experimentation with large populations of programs under controlled conditions.

The diversity present in the RSDIMU experiment was not enough to prevent significant correlation in the occurrence of conceptually-related errors, input-related errors, and eventual failures during testing. In order to prevent such correlation, measures beyond development and certification site diversity should be employed. New methods need to be developed to enforce diversity on development and testing if the reliability of N-Version structures is to be improved. Diversity of development and testing methodologies are promising .

We have developed a generalized interactive checker for asserting the correctness of a program on a given input. The generalized interactive checkers reduce the correctness

determination to testing the equality of two sets generated from the input and output under test. We have demonstrated the use of this method for typical flight domain examples. Generalized interactive checkers are promising for the implementation of acceptance checks for recovery blocks. Moreover, generalized interactive checkers is applicable to the validation of conventional (single version) high reliability software.

If it is possible to develop an independent acceptance check with a reasonable reliability -- such as those described above -- then the decision of whether to develop an N-Version structure or recovery block structure can be based upon the number of redundant components which are allowed by the available resources. For a large number of components it is better to build N-Version structures, while for a small number of components it is better to build recovery block structures. In the RSDIMU experiment we determined that for a three version structure, the acceptance check reliability need only be 80.5% or better to warrant the use of a recovery block structure. This result indicates that future research should be concentrated on development of independent acceptance checks.

Acknowledgments

The authors wish to express their gratitude to the following individuals who participated in this work. Thanks to Dr. Dave Eckhardt for his help in reliability modelling and general guidance in this research and to Dr. Jay Lala for his management of the subcontract. Gene Itkis participated in the development of acceptance checks. Cheryl Stubbs assisted us in reliability modelling and comparison of N-Version and Recovery Block software structures. Prof. Mladen Vouk of NCSU assisted in the diagnosis and identification of software errors in the RSDIMU programs.

6. REFERENCES

Babai, S., *Moran, Arthur-Merlin Games: A Randomized Proof System, and Hierarchy of Complexity Classes*, to appear in Journal of Comp. Sci. and Sys.

Blum, M. and Raghavan, P., *Program Correctness: Can One Test for It?*, IBM Research Report RC14038, September 1988.

Brilliant, S., Knight, J., and Leveson, N., *Analysis of Faults in an N-Version Software Experiment*, IEEE Trans. on Soft. Eng., Vol. 16, No. 2, February 1990.

Eckhardt, D. and Lee, L., *A Theoretical Basis for the Analysis of Software Subject to Coincident Errors*, IEEE Trans. on Soft. Eng., Vol. SE-11, No. 12, December 1985.

Eckhardt, D., Caglayan, A., Knight, J., Lee, L., McAllister, D., Vouk, M. and Kelly, J., *An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*, Submitted to IEEE Trans. on Soft. Eng.

Goldwasser, S., Micali, S., and Rackoff, C., *The Knowledge Complexity of Interactive Proof Systems*, Proc. of 27th FOCS, 1985.

Littlewood, B. and Miller, D., *Conceptual Modeling of Coincident Failures in Multiversion Software*, IEEE Trans. on Soft. Eng., Vol. 15, No. 12, December 1989.

Lorczak, P. and Caglayan, A., *A large Scale Second Generation Experiment in Multi-Version Software: Analysis of Software and Specification Errors*, Charles River Analytics Report No. R8903, January 1989.

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187492		

4. Title and Subtitle	5. Report Date
Software Reliability Experiments Data Analysis and Investigation	January 1991
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
J. Leslie Walker and Alper K. Caglayan	
	10. Work Unit No. 505-66-21
9. Performing Organization Name and Address	11. Contract or Grant No.
The Charles Stark Draper Laboratory Inc. Cambridge, MA 02139	NAS1-18061
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
NASA Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

This work was prepared by Charles River Analytics Inc., under Charles Stark Draper Laboratory Inc. subcontract no. 791. J. Leslie Walker and Alper K. Caglayan, Charles River Analytics Inc., Cambridge, Massachusetts.
Langley Technical Monitor: Dave E. Eckhardt, Jr.

16. Abstract

The objectives of this study are to investigate the fundamental reasons which cause independently developed software programs to fail dependently, and to examine fault-tolerant software structures which maximize reliability gain in the presence of such dependent failure behavior. We used 20 redundant programs from a software reliablility experiment to analyze the software errors causing coincident failures, to compare the reliability of N-version and recovery block structures composed of these programs, and to examine the impact of diversity on software reliability using subpopulations of these programs. The results indicate that both conceptually related and conceptually unrelated errors can cause coincident failures and that recovery block structures offer more reliability gain than N-version structures if acceptance checks that fail independently from the software components are available. We present a theory of general program checkers which have potential application for acceptance tests.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Fault tolerant software, software diversity, N-version, recovery block	Unclassified-Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	63	A04