[illegible]

[illegible] (1 + 3)

[illegible]

P 100

Semiannual Project Report Submitted to the

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Langley Research Center, Hampton, Va.

for research entitled

EXPERIMENTS IN FAULT TOLERANT SOFTWARE RELIABILITY

(NAG-1-667)

from

David F. McAllister, Co-Principal Investigator, Professor
K.C. Tai, Co-Principal Investigator, Professor
Mladen A. Vouk, Co-Principal Investigator, Assistant Professor

Department of Computer Science
North Carolina State University
Raleigh, N.C. 27695-8206
(919) 737-2858

Report Period
Beginning Date: April 1, 1987.
Ending Date: September 30, 1987.

(NASA-CR-181554) EXPERIMENTS IN FAULT TOLERANT SOFTWARE RELIABILITY Semiannual Progress Report, 1 Apr. - 30 Sep. 1987 (North Carolina State Univ.) 100 p Avail: NTIS HC A05/MF A01 CSCL 09B G3/61

N88-13863
--THRU--
N88-13866
Unclas
0111679

Table of Contents

Project Progress Summary

In this project we proposed to investigate a number of experimental and theoretical issues associated with the practical use of fault-tolerant software (FTS). In the period reported here we have worked on the following:

- We evaluated the reliability of voting in fault-tolerant software system for small output spaces,
- We investigated effectiveness of back-to-back testing process.
- Version 3.0 of RSDIMU-ATS, a semi-automatic test bed for certification testing of RSDIMU software developed in a previous experiment, was prepared and distributed for a multi-university N-version programming certification effort. Certification of the functionally equivalent components was completed during the summer 87.
- We continued studying software reliability estimation methods based on non-random sampling.
- We continued investigating existing and worked on formulation of new fault-tolerance models.

This report describes the results obtained in the period April 1, 1987 to September 30, 1987.

1. General Project Description

In the period 1985-87 NASA funded a multi-university experiment to develop 20 functionally equivalent software versions which are to be used to determine the reliability gains of several common fault-tolerant software systems. These include the two common methods of N-version programming and recovery-block [Ran75, Avi84] and hybrid schemes such as the consensus recovery block technique [Sco87]. Our main goal was to investigate the reliability of N-version programming which requires the comparison of outputs of several functionally equivalent software components to determine correctness. An important related issue is how to test these components for reliability and capitalize on the fact that they are functionally equivalent.

Although experimenters have shown that existing fault-tolerant software (FTS) techniques can achieve an improvement in reliability over non-fault-tolerant software systems, it has also been shown that failure dependence among FTS system components may not be negligible in the context of current software development and testing techniques [Nag82, Sco84, Nag84, Vou85, Kni86, Kel86]. Correlated coincidental component failures may be disastrous in current FTS approaches and can seriously undermine any reliability gains offered by the fault-tolerance mechanisms [e.g. Sco83a, Sco84, Avi84, Eck85, Vou86a]. Hence it is important to detect and eliminate them as early as possible in a FTS life-cycle.

This project is concerned with experimental and theoretical investigation of FTS problems. A four university effort to build and verify 20 functionally equivalent software components implementing an abstraction of a redundant strapped down inertial measurement unit (RSDIMU) was completed this summer. Analysis of the results of the experiment is underway. In connection with this effort it was necessary to research software development and testing techniques, and FTS voting models in order to identify characteristics that are relevant to development of redundant software components intended for FTS. This report describes the results obtained in the period April 1, 1987 to September 30, 1987.

In the period reported here we have worked on the following:

- We evaluated the reliability of voting in fault-tolerant software system for small output spaces.
- We investigated effectiveness of back-to-back testing process.
- Version 3.0 of RSDIMU-ATS, a semi-automatic test bed for certification testing of RSDIMU software developed in a previous experiment, was prepared and distributed for field testing in a multi-university N-version programming effort. Recertification of the functionally equivalent components was completed during the summer 87.
- We continued studying software reliability estimation methods based on non-random sampling.
- We continued investigating existing and worked on formulation of new fault-tolerance models.

2. Results

2.1 Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces

Under a voting strategy in a fault-tolerant software system there is a difference between correctness and agreement. An independent N-version programming reliability model is proposed for treating small output spaces which distinguishes between correctness and agreement. System reliability is investigated using analytical relationships and simulation. A consensus majority voting strategy is proposed and its performance is analyzed and compared with other voting strategies. Consensus majority strategy automatically adapts the voting to different component reliability and output space cardinality characteristics. It is shown that absolute majority voting strategy provides a lower bound on the reliability provided by the consensus majority, and 2-of-n voting strategy an upper bound. If r is the cardinality of the output space it is proved that 1/r is a lower bound on the average reliability of fault-tolerant system components below which the system reliability begins to deteriorate as more versions are added.

Report on the execution time distributions is in Appendix I of this report.

2.2 Effectiveness of Back-to-Back Testing

Results are presented of an experiment in back-to-back testing using the functionally equivalent software versions mentioned earlier in this report. These versions were used to evaluate use of back-to-back testing in initial stages of the software testing process where a difference among outputs signals a potential system fault. It is shown that a significant increase in the probability of detecting failures can be achieved using back-to-back testing provided failure correlation between versions is sufficiently small. It was found that in some cases back-to-back testing may provide a failure detection gain even with relatively high correlation. Three models of back-to-back testing process of multiple functionally equivalent software versions are described and compared. Two models treat the case where version failures are independent while the third model describes the more realistic case where there is correlation among the failures. The cost of the approach over single version development is discussed briefly.

Report on back-to-back testing is in Appendix II of this report.

2.3 RSDIMU Certification Testing

RSDIMU Acceptance Testing System (RSDIMU-ATS) version 3.0 was released to help test and analyze multiversion RSDIMU procedures during the certification phase of the experiment in the summer 1987. This system was released for restricted use by sites involved in the NASA-LaRC fault-tolerant software experiment. RSDIMU-ATS is intended for use in a UNIX environment and may need to be modified if UNIX-like, or non-VAX systems are used. Detailed release notes for version 3.0 are given in Appendix III.

Recertification of of the programs assigned to NCSU was completed during the summer 1987.

2.4 Other Work in Progress

Investigations in progress are continued study of the effectiveness of back-to-back testing, and a performance study of multistage fault-tolerant software systems. Reports on both activities are expected in the next report period.

Bibliography

[Avi84] Avizienis and J.P. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, pp. 67-80, 1984 .

[Bri86] S. Brilliant and J.C. Knight, "Testing Software Using Multiple Versions", University of Virginia, Department of Computer Science, Report No. RM-86-07, 1986.

[Dur84] J.W. Duran and S.C. Ntafos, "An evaluation of random testing", IEEE Trans. Soft. Eng., Vol. SE-10, 438-444, 1984

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Ehr85] W. Ehrenberger, "Statistical Testing of Real Time Software", in "Verification and Validation of Real Time Software", ed. W.J. Quirk, Springer-Verlag, 147-178, 1985.

[How87] W.E. Howden, "Functional Program Testing and Analysis", McGraw-Hill Book Co., 1987.

[Kel86] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "Early Results from the Second Generation Multi-Version Software Experiment", submitted for publication, 1986.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multiversion Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[McA87] D.F. McAllister, C.E. Sun, and M.A. Vouk, "Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces", North Carolina State University, Department of Computer Science, Technical Report, TR-87-16, submitted for publication, 1987.

[Nag82] P.M. Nagel and J.A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling", BSC-40366, Boeing, Seattle, Wa., 1982

[Nag84] P.M. Nagel, F.W. Scholz and J.A. Skrivan, "Software Reliability: Additional Investigation into Modeling with Replicated Experiments", NASA CR172378, Boeing, Seattle, Wa., 1984

[Ran75] B. Randell, "System structure for software fault-tolerance", IEEE Trans. Soft. Eng., Vol. SE-1, 220-232, 1975.

[Sag86] F. Saglietti and W. Ehrenberger, "Software Diversity -- Some Considerations about Benefits and its Limitations", Proc. IFAC SAFECOMP '86, 27-34, 1986.

[Sco83a] R.K. Scott, "Data Domain Modeling of Fault Tolerant Software Reliability", Ph.D. Dissertation, North Carolina State University, Raleigh, North Carolina, 1983

[Sco83b] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", Proc. IEEE 14th Fault-Tolerant Computing Symposium, pp. 102-107, 1983

[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984

[Sco87] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling", IEEE Trans. Software Eng., Vol SE-13, 582-592, 1987.

[Vou85] M.A. Vouk, D.F. McAllister, K.C. Tai, "Identification of correlated failures of fault-tolerant software systems", in Proc. COMPSAC 85, 437-444, 1985.

[Vou86a] M.A. Vouk, D.F. McAllister, K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-tolerant Software", Proc. Workshop on

Software Testing, Banff, Canada, IEEE CS Press, 74-81, July 1986.

[Vou86b] M.A. Vouk, M.L. Helsabeck, K.C. Tai, and D.F. McAllister, "On Testing of Functionally Equivalent Components of Fault-Tolerant Software", Proc. COMPSAC 86, 414-419, 1986.

[Vou86c] M.A. Vouk, and D.F. McAllister, "A Proposed Methodology for The Development of Fault-Tolerant Software", North Carolina State University, Department of Computer Science, Technical Report TR-86-11, 1986.

[Vou87] M.A. Vouk, D.F. McAllister, D.E. Eckhardt, A. Caglayan, and J.P.J. Kelly, "Effectiveness of Back-to-Back Testing", North Carolina State University, Department of Computer Science, Technical Report TR-86-08, 1987, submitted for publication.

Appendix I

N88-13864

COMPUTER STUDIES

TECHNICAL REPORT

Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces

McAllister, David F.
Sun, Chien-En
Vouk, Mladen A.

TR-87-16

North Carolina State University

Raleigh, N. C. 27650

Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces

David F. McAllister
Chien-En Sun
Mladen A. Vouk

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206

Abstract

Under a voting strategy in a fault-tolerant software system there is a difference between correctness and agreement. An independent N-version programming reliability model is proposed for treating small output spaces which distinguishes between correctness and agreement. System reliability is investigated using analytical relationships and simulation. A consensus majority voting strategy is proposed and its performance is analysed and compared with other voting strategies. Consensus majority strategy automatically adapts the voting to different component reliability and output space cardinality characteristics. It is shown that absolute majority voting strategy provides a lower bound on the reliability provided by the consensus majority, and 2-of-n voting strategy an upper bound. If r is the cardinality of the output space it is proved that $1/r$ is a lower bound on the average reliability of fault-tolerant system components below which the system reliability begins to deteriorate as more versions are added.

*Research supported in part by NASA Grant No. NAG-1-667

I. Introduction

Recent experiments with multiversion software have demonstrated the fact that identical and incorrect answers can occur with perhaps higher frequency than is expected [Sco84, Kni86, Vou85,86] particularly when small output spaces are involved. For example, if the output space is binary, {0,1}, then all incorrect responses must agree. Such phenomena make the fault-tolerant techniques of N-version programming [Avi77, Avi84] and Consensus Recovery Block [Sco83a,b,84,87] more likely to fail during the voting process since the voter may not be able to distinguish between correct and incorrect responses. In the past models of fault-tolerant reliability have equated output agreement with correctness (e.g. [Sco83a,b, Sco87]) which is inadequate for complete modeling of such an environment. In this paper we distinguish between agreement and correctness and develop a reliability model of a voting environment which can be used to determine the number of versions required as a function of the cardinality of the output space.

II. Voting Strategies in N-Version Programming

In an ***m-of-n fault-tolerant software (FTS) system*** the number of functionally equivalent independently developed versions is n, and m is the ***agreement number*** or the number of matching outputs which the voting or adjudication algorithm requires for system success. In the past, because of cost restrictions, n was rarely larger than 3 and m was traditionally chosen as Ceiling[(n+1)/2] which we will call ***absolute majority voting.*** In [Sco87] Scott, Gault and McAllister show that if the output space is large and with true statistical independence of FTS versions, there is no need to choose m > 2 regardless of the size of n although considerable reliability gains occur with larger n. We will use the term ***2-of-n voting*** for this case.

With small output spaces we suggest that a third voting technique be considered, which we will call ***consensus majority voting*** . To motivate this technique consider the following scenario. Suppose we have n = 11 versions and output space cardinality of 3. Let a vector (i,j,k) represent the frequencies of the three possible output states (i + j + k = n) and let the first component, i, represent the frequency of the correct output. In this case, absolute majority is 6, but vectors (5,3,3), (5,4,2), or (5,2,4) may represent likely events which will be declared a system failure under absolute majority voting. Furthermore, the vectors (4,4,3) and (4,3,4) are the only cases in which a correct answer occurs when exactly four versions agree. But if three is chosen as the agreement number, there always exists another output on which more than three versions agree. In such cases an obvious strategy is to choose the output with the largest frequency, if such exists.

When there is more than one choice, as in this example when the output state frequency vector is (5,5,1) or (5,1,5), and if choosing a wrong answer or having no answer has the same impact on the system, then choosing one result with five identical outputs at random is a better strategy (on the average) than declaring system failure. In this example then there is a 50 percent chance that the correct output will be selected when this formal strategy is used.

In consensus majority voting the voter uses the following algorithm to select the "correct" answer:

- If there is an absolute majority agreement ($\geq$ Ceiling[(n+1)/2]) then this answer is chosen as the " correct" answer.
- If there is a unique maximum agreement, but this number of agreeing versions is less than Ceiling[(n+1)/2], then this answer is chosen as the "correct" one.
- If there is a tie in the maximum agreement number then one set is chosen at random and the answer associated with this set is chosen as the "correct" one.

We discuss this strategy further in the following sections and compare it with 2-of-n and absolute majority voting. We will first develop the mathematics for treating the difference between agreement and correctness.

III. The Correctness Factor

We first define a correctness factor c_i which is the probability that an output is correct given that i versions agree. If $Pr_c(i)$ is the probability that i versions are correct and $Pr_a(i)$ is the probability that i versions agree we have

$$\begin{aligned} c_i &= Pr\{\text{output is correct} \mid \text{i versions agree}\} \\ &= Pr\{\text{i versions are correct}\} / Pr\{\text{i versions agree}\} \\ &= Pr_c(i) / Pr_a(i) \qquad (1) \end{aligned}$$

Now

$$\begin{aligned} Pr\{\text{i versions agree}\} = \; & Pr\{\text{i versions agree and are correct}\} + \\ & Pr\{\text{i versions agree and are incorrect}\} \qquad (2) \end{aligned}$$

Also, using "correct" and "incorrect" to mean "output is correct" and "output is incorrect" respectively

$$\Pr\{\text{incorrect} \mid i \text{ versions agree}\} + \Pr\{\text{correct} \mid i \text{ versions agree}\} = 1$$

Hence,

$$\Pr\{\text{incorrect} \mid i \text{ versions agree}\} = 1 - c_i$$

Using the data domain approach of Scott et. al. [Sco83a,b], the reliability of an m-of-n system which we denote by $R_{m|n}$ becomes

$$R_{m|n} = \sum_{i=m}^{n} Pr_c(i) = \sum_{i=m}^{n} c_i \, Pr_a(i) \qquad (3)$$

We expect that the sequence

$$C = \{c_i \mid i = 2,\ldots,n\}$$

is nondecreasing, i.e. that the chance of an output being incorrect does not increase as the number of versions which agree increases. In particular, it is clear that

$$R_{m|n} \le \max\{c_i\} \sum_{i=m}^{n} Pr_a(i) \le \max\{c_i\}$$

and hence if, for all i, $c_i < 1$ then $R_{m|n} < 1$.

For tractability we will assume that all software versions have the same reliability or probability of obtaining the correct answer for a given input. Let this reliability be p. Then we have

$$Pr_c(i) = {}_nC_i \, p^i (1-p)^{n-i}$$

where ${}_nC_i$ denotes the number of combinations of n items taken i at a time. It follows that

$$Pr_a(i) = Pr_c(i) + {}_nC_i (1-p)^i p^{n-i} \qquad (4)$$

In the following section we consider the impact of small output spaces on a voting strategy.

Table 3.1 gives correctness factors as a function of version reliabilities when the output space is binary. The correctness factors converge to 1 only for $p>1/r$ where r is the size of the output space. (Indeed the sequence is decreasing for $p < 1/r$) .This is not an accident as we will show in the following section.

IV. Small Output Spaces

Let the output space have cardinality r and assume as above that all versions have the same reliability p. We further assume, for tractability, that the probability of failure of a version is independent of the failure of any other. Assume a labeling of the outputs 1,2 ..., r such that output 1 is the correct one and occurs n_1 times. Let q_i , $i \geq 2$ denote the probability that the incorrect output i will occur n_i times where

$$n_1 + n_2 + \ldots + n_k = n$$

and

$$p + q_2 + q_3 + \ldots + q_k = 1.$$

Then the probability that the correct output 1 will occur n_1 times is

$$P_c(n_1) = \sum_{\sum_{i=2}^{n} n_i = n - n_1} \frac{n!}{n_1! n_2! \ldots n_r!} p^{n_1} q_2^{n_2} \ldots q_r^{n_r} \qquad (5)$$

where i >1. The reliability of an m-of-n system becomes

$$R_{min} = \sum_{n_1 = m}^{n} \left[\sum_{\sum_{i=2}^{n} n_i = n - n_1} \frac{n!}{n_1! n_2! \ldots n_r!} p^{n_1} q_2^{n_2} \ldots q_r^{n_r} \right] \qquad (6)$$

Equation (6) does not allow for multiple incorrect outputs, however. Let M $(i;n_j)$ denote that i replaces n_j in equation (5). That is,

$$M(i\,;n_j) = \sum_{\sum n_k = n-i} \frac{n!}{n_1!\ldots n_{j-1}!\; i!\; \ldots n_r!}\, p^{n_1} q_2^{n_2} \ldots q_{j-1}^{n_{j-1}} q_j^{i} \ldots q_r^{n_r} \qquad (7)$$

It follows that the correctness factor c_i then becomes

$$c_i = \frac{M(i\,;n_1)}{\sum_{j=1}^{r} M(i\,;n_j)} \qquad (8)$$

where i lies between Ceiling[(n+1)/2] and n. The terms $M(i;n_j)$ are the probabilities of exactly i identical incorrect outputs where $2 \le j \le r$. We note that the expression becomes considerably more complicated when i lies between Floor [(n+r-1)/r] and Ceiling [(n+1)/2] since the $M(i\,;n_j)$'s may have terms in common, i.e., it is possible to have more than one output occurring more than i times. For example, in a case in which r=3 and n=11 it is possible that the correct output and one incorrect output can each occur 5 times for the same input. In this case the denominator of equation (8) overestimates the correct result. We assume henceforth that $i \ge$ Ceiling [(n+1)/2].

For tractability we also assume the occurrence of each incorrect output has the same probability q, and $(r-1)q = 1-p$. (This assumption is also 'best case' in the sense that different probabilities tend to effectively reduce the output space requiring higher version reliability. We will discuss this further in the last section.) Equation (7) then becomes

$$M(i\,;n_j) = \sum_{\sum n_k = n-i} \frac{n!}{n_1!\ldots n_{j-1}!\; i!\; \ldots n_r!}\, p^{n_1} q^{n-n_1} \qquad (9)$$

Then we have

$$M(i;n_2) = M(i;n_3) = \ldots = M(i;n_k)$$

and equation (8) becomes

$$c_i = \frac{M(i\,;n_1)}{M(i\,;n_1) + (r-1)M(i\,;n_2)} \qquad (10)$$

From [Tri82], the marginal probability function of $M(i;n_j)$ is given by

$$M(i;n_j) = {}_nC_i\, q^i (1-q)^{n-i} \tag{11}$$

When $i = n_1$ the equation becomes

$$M(i;n_1) = {}_nC_i\, p^i (1-p)^{n-i} \tag{12}$$

Substituting (11) and (12) into (10) gives

$$c_i = \frac{[{}_nC_i\, p^i (1-p)^{n-i}]}{[{}_nC_i\, p^i (1-p)^{n-i} + (r-1)\,{}_nC_i\, q^i (1-q)^{n-i}]} \tag{13}$$

From our comments above and equation (3), it follows that we cannot have R_{min} converging to 1 unless the sequence $\{c_i\}$ converges to 1. In the following theorem we show that a necessary and sufficient condition that $\lim\{c_i\} = 1$ is that $p > 1/r$.

Theorem 4.1:

The sequence $\{c_i\}$ is increasing and $\lim \{c_i\} = 1$ (as $n \to \infty$) if and only if $p > 1/r$ $(r \geq 2)$.

Proof:

The sequence $\{c_i\}$ shown in equation (13) can be simplified to

$$c_i = \frac{1}{1 + (r-1)\,(q/p)^i\,[\,(1-q)/(1-p)]^{n-i}} \tag{14}$$

In equation (14), let

$$Q_i = (q/p)^i\,[(1-q)/(1-p)]^{n-i} \tag{15}$$

then equation (14) becomes

$$c_i = 1/[1+(r-1)Q_i] .$$

Substitution of $q=(1-p)/(r-1)$ into equation (15) gives

$$Q_i = \frac{(1-p)^{2i-n}(r-2+p)^{n-i}}{(r-1)^n p^i} \qquad (16)$$

We note that $\{c_i\}$ is an increasing sequence and its limit is one if and only if $\{Q_i\}$ is a decreasing sequence converging to zero. For the sequence $\{Q_i\}$ to be monotone we must have $(Q_{i+1}/Q_i) < 1$. Since,

$$\frac{Q_{i+1}}{Q_i} = \frac{(1-p)^2}{(r-2+p)p} = \frac{1-2p+p^2}{p(r-2)+p^2} < 1$$

or

$$p(r-2)+p^2 > 1-2p+p^2$$

or

$$(r-2)p > 1-2p$$

or

$$rp > 1$$

Hence, we must have

$$p > 1/r .$$

This proves the necessity.

Since the boundary reliability p is larger than $1/r$, let Δ denote a positive number such that

$$p = 1/r + \Delta$$

Substituting $(1/r + \Delta)$ into equation (16), we have

$$Q_i = \frac{\left(1 - \frac{1}{r} - \Delta\right)^{2i-n}\left(r - 2 + \frac{1}{r} + \Delta\right)^{n-i}}{(r-1)^n\left(\frac{1}{r} + \Delta\right)^i} \tag{17}$$

Then

$$\frac{Q_{i+1}}{Q_i} = \frac{\left(1 - \frac{1}{r} - \Delta\right)^2}{\left(r - 2 + \frac{1}{r} + \Delta\right)\left(\frac{1}{r} + \Delta\right)}$$

Because r is an integer and larger than or equal to two,

$$\left(\frac{1}{r} + \Delta\right)\left(r - 2 + \frac{1}{r} + \Delta\right) - \left(1 - \frac{1}{r} - \Delta\right)^2 = r + r\Delta - 1 > 1$$

Therefore, $\{Q_i\}$ is a decreasing sequence, and hence $\{c_i\}$ is an increasing sequence. When $i = n$, Q_n becomes

$$Q_n = \left(\frac{(r-1) - r\Delta}{(r-1) + (r-1)r\Delta}\right)^n$$

Because the fraction is less than one, Q_n approaches zero as n approaches infinity. Therefore, the limit of $\{c_i\}$ is one when p is larger than 1/r. **This proves sufficiency** and completes the proof of Theorem 4.1.

The following theorem gives sufficient conditions that

$$\lim_{n \to \infty} R_{min} = 1$$

<u>Theorem 4.2:</u>

Let the output space have cardinality r and assume all components are independent and have the same reliability p. Further assume unique correct outputs. Then the following are sufficient conditions for

$$\lim_{n \to \infty} R_{min} = 1$$

(1) $p > 1/r$.

(2) The agreement number m is equal to Floor[(n+r-1)/r]. If Floor[(n+r-1)/r] is zero, m is set to two. (We note that if n becomes arbitrarily large then m must also).

(3) When a version fails, the probability of occurrence of any incorrect output is q, where $q=(1-p)/(r-1)$.

Proof:

The probability that the j^{th} incorrect item within the output space is generated by i versions is $M(i\,;n_j)$. From equation (11), the marginal probability of $M(i\,;n_j)$ is

$$M(i\,;n_j) = {}_nC_i\, q_j^i\,(1-q_j)^{n-i}$$

where q_j is the probability that the jth incorrect output is generated. When j is not smaller than the majority this incorrect output will be voted as correct in N-version programming. But it may or may not be voted as the correct answer when j is a number between m and Ceiling[(n+1)/2]. Depending on the voting strategy the probability that the j^{th} incorrect item may be chosen as the correct answer under m-of-n voting algorithm is no larger than

$$H_j = \sum_{i=m}^{n} {}_nC_i\, q_j^i\,(1-q_j)^{n-i}$$

In [Aik55], the above binomial formula is approximated using the following expression

$$H_j = \frac{1}{\sqrt{2\pi}}\int_{k_1}^{k_2} e^{-\frac{i^2}{2}}\,di + \frac{1-2q_j}{6\rho\sqrt{2\pi}}\,[(1-k^2)e^{-\frac{k^2}{2}}]_{k_1}^{k_2} + \omega \qquad (18)$$

where

$$k_1 = \frac{m - nq_j - \frac{1}{2}}{\sqrt{nq_j\,(1-q_j)}} \qquad (19)$$

$$k_2 = \frac{m - nq_j + \frac{1}{2}}{\sqrt{nq_j\,(1-q_j)}} \qquad (20)$$

$$\rho = \sqrt{nq_j(1-q_j)} \tag{21}$$

and the error term ω satisfies the inequality

$$|\omega| < \frac{.13 + .18|1-2p|}{\rho^2} + e^{-\frac{3\rho}{2}}$$

Let $p = (1/r + \Delta)$, where Δ is a positive number.

Because $q_j = q = (1-p)/(r-1)$,

we have $q_j = q = (1/r)-\partial$, where $\partial = \Delta/(r-1)$.

Substituting $q_j = (1/r)-\partial$ into equation (19) it becomes

$$k_1 = \frac{m - n\left(\frac{1}{r} - \partial\right) - \frac{1}{2}}{\sqrt{n\left(\frac{1}{r} - \partial\right)\left(1 - \frac{1}{r} + \partial\right)}}$$

Since $m \geq n/r$, (m=Floor[(n+r-1)/r]), we have

$$k_1 \geq k' = \frac{\partial n - \frac{1}{2}}{\sqrt{n\left(\frac{1}{r} - \partial\right)\left(1 - \frac{1}{r} + \partial\right)}}$$

$$= \frac{\partial n - \frac{1}{2}}{\beta\sqrt{n}}$$

$$= \frac{\partial}{\beta}\sqrt{n} - \frac{1}{2\beta\sqrt{n}}$$

where

$$\beta = \sqrt{\left(\frac{1}{r} - \partial\right)\left(1 - \frac{1}{r} + \partial\right)}$$

Obviously, when n approaches infinity, k' becomes arbitrarily large. Since k' is no larger than k_1 and k_2 ($k_1 < k_2$),

$$\lim_{n \to \infty} k_1 = \infty$$

and

$$\lim_{n \to \infty} k_2 = \infty$$

When n approaches infinity, ρ becomes arbitrarily large. The error term ω in equation (18) approaches zero, and the limit can be written as

$$\lim_{n \to \infty} H_j = \lim_{k_1, k_2 \to \infty} \left[\frac{1}{\sqrt{2\pi}} \int_{k_1}^{k_2} e^{-\frac{i^2}{2}} di + \frac{1 - 2q_j}{6\rho\sqrt{2\pi}} [(1 - k^2)e^{-\frac{k^2}{2}}]_{k_1}^{k_2} \right] \qquad (22)$$

The integral in the above equation is the accumulation function of the standard normal distribution. Since k_1 and k_2 approach infinity the limit is zero.

In treating the limit of the second part, let

$$(H_j)_2 = (1 - k)e^{-\frac{k^2}{2}} \qquad (23)$$

Applying L'Hopital's rule and differentiating both numerator and denominator with respect to k gives

$$(H_j)_2' = \frac{1 - 2k}{ke^{\frac{k^2}{2}}}$$

$$(H_j)_2'' = \frac{1}{e^{\frac{k^2}{2}} + k^2 e^{\frac{k^2}{2}}}$$

Obviously, the numerator of the above equation approaches zero when k becomes very large. Therefore, we have

$$\lim_{k\to\infty}(H_j)_2 = \lim_{k\to\infty}(H_j)''_2 = 0.$$

The second part of equation (22) consists of two items

$$(1-k_1^2)\,e^{-\frac{k_1^2}{2}} \text{ and } (1-k_2^2)\,e^{-\frac{k_2^2}{2}}$$

Since the values of both approach zero as k_1 and k_2 become arbitrarily large, the difference between the two approaches zero. This establishes that as n becomes arbitrarily large the value computed by equation (18) approaches zero.

Since the occurrence of each incorrect output has the same probability of appearing, the unreliability of this m-of-n software system ($F_{m|n}$) = 1 - $R_{m|n}$ satisfies

$$F_{m|n} < (r-1)\sum_{i=m}^{n} {}_nC_i\, q^i\,(1-q)^{n-i} \qquad (24)$$

Therefore, when n approaches infinity the right side of the inequality is zero. Because $F_{m|n}$ should be non-negative, it also approaches zero. The reliability of the system approaches one. **This completes the proof** of Theorem 4.2.

V. Examples

In this section we present numerical examples which illustrate the effect on the system reliability of different version reliabilities, different output space cardinality, and different voting strategies.

In Table 5.1 and Figure 5.1 we show results obtained using equation (6) with m=Ceiling[(n+1)/2] and r=2. This is the classical majority voting approach with a binary output space. The boundary version reliability in this case is 1/r = 0.5. The three rows in the middle of Table 5.1 show that the version reliability must be larger than the boundary version reliability in order to improve the performance of the system. Figure 5.1 shows that with a fixed version reliability larger than 0.5

system reliability increases when more versions are added. This, of course, is in agreement with findings of Eckhardt and Lee [Eck85] who also studied the absolute majority voting process.

Again using equation (6) if we assume an output space cardinality of r=3 then Table 5.2 and Figure 5.2 show the effect on system reliability of varying the version reliabilities and the number of versions m for the consensus majority voting strategy. The minimal agreement number is Floor[(n+r-1)/r] = Floor[(n+2)/3]. The average boundary reliability of the versions is 1/r = 1/3. Below this reliability value the sequence of correctness factors $\{c_i\}$ decreases and the system reliability deteriorates as more versions are added. All versions are assumed to have the same reliability, and all failure states (j=2,3) the same probability (1-p)/(1-r)=(1-p)/2 of being excited.

Table 5.3 and Figure 5.3 summarize the effect of the 2-of-n voting strategy for different numbers of components. The reliability was computed using equations (3) and (4). The agreement number is m=2, and it is assumed that the output space cardinality is infinite. As the number of components in the system increases, the reliabilities rapidly decrease, but here this effect is related to the number of components involved in the voting rather than the output space cardinality. Of course, unless $c_2 = c_3 = ... = c_n = 1$ this voting strategy can lead to disaster.

The relationship between r and voting strategies is illustrated in Figure 5.4 for n=15. It is important to note that both the absolute majority and the 2-of-n are effectively output space insensitive and can lead to ambiguous or nonunique results. For odd n the former behaves as if r=2 since for absolute majority voting the agreement number is Ceiling[(n+1)/2] which is equivalent to letting r=2 in the agreement number equation for consensus majority voting, Floor[(n+r-1)/2]. For even n Ceiling[(n+1)/2]>Floor[(n+2-1)/2]. Therefore from equations (3) and (4) it follows that given r=2 for even n the reliability of the system using absolute majority voting will be lower than when consensus majority is used, while for odd n it will be equal to it. The 2-of-n voting behaves as if r=infinity since for infinite r the consensus majority agreement number reduces to 2 (see Theorem 4.2). The consensus majority voting is r sensitive and therefore will perform better than absolute majority voting for r>2 since Floor[(n+r-1)/r]$\leq$Ceiling[(n+1)/2]. The absolute majority represents a lower limit of the consensus majority voting with r=2, while the 2-of-n is an upper limit.

Dependence of the system reliability on the output space cardinality is further illustrated in Figure 5.5 for consensus majority with n=5 and n=15. Failure state probabilities are the same for all j=2..r incorrect outputs. We note that the asymptotic system reliability ($r=\infty$) corresponds to 2-of-n voting approach, while the r=2 point corresponds to the absolute majority voting. Equations (3) to

(6) with consensus voting and simulation were used to compute the data points shown in Figures 5.4 and 5.5.

VI. Simulation

The relationships given in section IV were derived by assuming that the probabilities of all failure states are equally likely. In practice this may not be true. In fact, it is quite possible that one of the failure states j, $2 \leq j \leq r$ is preferentially excited because of very high visibility (under given input conditions) of the fault(s)/errors mapping into it. In the extreme, even in a large output space, this leads to the behaviour of the fault-tolerant system as if the output space cardinality is small, i.e. highly visible errors force a partitioning on the output space into equivalence classes which effectively reduces the output space cardinality.

Another approximation that was made is the assumption that all the components have the same reliability. In practice a range of reliabilities around a mean value, p, would be expected. To study the influence of the scatter of individual component reliabilities and of the scatter in the probabilities of incorrect outputs, and to check on analytical solutions we used simulation.

The model we have used is illustrated in Figure 6.1. A single component i is assumed to exhibit a probability $q_i = (1-p_i)$ of failing. We do not separately model different errors contributing to this average failure rate, and we assume that all the components exhibit mutual independence with respect to the probability of failure. For each simulated input the component state (failed, not_failed) is chosen randomly with the probability, q_i, assigned to that component. If the final component state is a failure state the actual output state j, one of the (r-1) incorrect outputs, is selected randomly with the conditional probability P(j|i-failed) associated with that output. The process is repeated for all n components. The final states of the components are then voted using the absolute majority, consensus majority and 2-of-n strategies.

Simulation of systems described by the equations given in section IV yielded results that coincided with the computations obtained using analytic solutions to within the confidence interval of the simulation runs.

The influence of the scatter in component reliabilities is illustrated in Figures 6.2 and 6.3. The standard deviation of component reliability, the square root of $\sigma_p^2 = \Sigma[(p_i - p)^2/(n-1)]$, where $p = \Sigma p_i/n$, for i=1..n, is used to measure the dispersion of the component reliability values. In Figure 6.2 we plot system reliability against the standard deviation of the component reliability using n=5 with r=4 and p=0.95. It was assumed that each incorrect output state j has an equal probability (1-p)/(r-1) of being selected. In Figure 6.3 we have n=5, r=4 and p=0.623. Each pair of points (majority-absolute) shown in Figures 6.2 and 6.3 was obtained from a separate 100,000 case simulation run.

From the figures we see that the larger the standard deviation of the component reliabilities the more reliable the system. This confirms that from the point of view of component reliabilities the equations discussed in section IV provide a conservative estimate of the system behaviour since they use the same component reliability value for all components and imply a standard deviation of zero. Of course, if scatter is large enough then it may happen that one of the components is in fact more reliable than the system as a whole under some or all of the discussed voting strategies.

For example, given that n=5, r=4, and p=0.623 with σ_p=0.186 we can have p_1=0.759, p_2=0.522, p_3=0.357, p_4=0.819, p_5=0.658. These component reliability values give system reliability of 0.735 for absolute majority voting, and 0.851 for consensus majority voting. Clearly, component i=4 on its own is more reliabile than the 5-version system under absolute majority vote, and is marginally worse than the system under consensus majority voting. As another example consider a system composed of more reliable components. Let n=5, r=4, and p=0.95000 with σ_p=0.05333 which we can produce with p_1=0.96456, p_2=0.91313, p_3=0.87732, p_4=0.99999, p_5=0.99500. The resulting system reliability under absolute majority voting is 0.99953, and under consensus majority voting is 0.99979. Component i=4 is more reliable than the system under either of the strategies. Values in first example were computed using 2,000,000 case simulations giving a 95% confidence range about obtained system reliabilities of about ± 0.00003. In the second example we used a 10,000,000 case simulation giving 95% confidence limits of about ± 0.000013 about the reported values.

Therefore, if all the components are nearly equally reliable, i.e. scatter is small, then using equations given in section IV to predict system reliability will provide a conservative estimate of this reliability. But if the scatter of the reliabilities is large it means that at least one of the components is much more reliable than the average over all the components, and it may happen that there is at least one component which is more reliable than the N-version system. In such a situation, one should either reduce the system by discarding the most unreliable component(s), or perhaps employ modified voting strategies. To illustrate this consider the following.

Returning to the second example above and discarding component $p_3=0.87732$ and keeping the other four, results in system reliability of 0.99635 under absolute majority voting and 0.99937 under consensus majority voting. By discarding p_3 and p_2 we obtain 0.99979 for both absolute and consensus majority voting. It is only when we reduce the system down to two components that the sytem reliability becomes larger than that of component 4. Simulations ran for 10,000,000 cases giving 95% confidence limits on system reliability of about $\pm$ 0.00001. Clearly, when working with high reliability components, it is very important to have a well balanced set of components of nearly equal reliability in order to achieve best results. A possible adaptive voting strategy would be given information about the individual component reliabilities and may, for example, attach more importance to answers from sets containing the most reliable component(s). This is a subject for future research.

Figure 6.4 illustrates the effect of variation in the conditional probabilities of selecting incorrect output states. If $q_{ij}=q_iP(j|i\text{-failed})$ represents the probability of selecting j^{th} incorrect output for i^{th} component, where $P(j|i\text{-failed})$ is the conditional probability that j^{th} state will be chosen, then $q_i=\Sigma q_{ij}=q_i\Sigma P(j|i\text{-failed})=1-p_i$, $j=2..r$. Let $P_i=\Sigma P(j|i\text{-failed})/(r-1)=1/(r-1)$ be the average conditional probability. Then the standard deviation shown in Figure 6.4 is the square root of $\sigma_P{}^2 = \Sigma [(P(j|i\text{-failed})-P_i)^2/(r-2)]$, where the sum is over $j = 2..r$ output states. Simulations were performed assuming that for individual component reliabilities $p_1=p_2=...=p_n=p$. We see that the larger the scatter of the conditional failure probabilities, assuming the same p for all components, the more the system behaviour tends towards that associated with the lower r values, i.e. toward that exhibited when absolute majority voting is employed. The curve for the latter is, of course, level since absolute majority voting is r insensitive (effective output space becomes binary, r=2). Simulation for each pair of points ran for 100,000 test cases.

As an example, let n=5, p=0.95 (all components), and r=4 with equal conditional failure probabilities for all incorrect outputs ($j=2,3,4$; $P(j|i\text{-failed})=1/3$). Then absolute majority voting has system reliability of 0.99884 and consensus majority voting has reliability 0.99994. On the other hand, if we let r=11 but P(2|i-failed)=0.99910 while for j=3..11 P(j|i-failed)=0.00010, then consensus majority reliability drops to 0.99884 which is equal to that obtained by absolute majority voting, and is equivalent to an effective reduction of the output space cardinality to r=2. Example values were obtained by simulation, and their standard deviation is 0.000025.

The above discussion leads to the conclusion that for conservative estimates we should use r=2 and an average p value. However, in practical applications use of consensus majority voting is recommended since it provides automatic adaptation of the voting strategy to the component reliability and output space characteristics. In the lower limit the reliability provided by consensus majority is never worse than the absolute majority voting, while in the upper limit it is equivalent to the 2-of-n voting strategy.

VII. Conclusions

We have analyzed fault-tolerant software systems using N-Version Programming and different voting algorithms assuming output spaces with small cardinality and version failure independence. We have proposed an alternative voting strategy which we call consensus majority voting to treat cases when there may be agreement among incorrect outputs, a case which can occur with small output spaces. Consensus majority voting provides automatic adaptation of the voting strategy to varying component reliability and output space characteristics. We show that if r is the cardinality of the output space then 1/r is a lower bound on the average reliability of fault-tolerant system versions below which system reliability begins to deteriorate as more versions are added.

VIII. References

[Aik55] H.H. Aiken et. al, "Tables of the Cumulative Binomial Probability Distribution", Harvard University Press, Mass., 1955.

[Avi77] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution", Proc. COMPSAC 77, 149-155, 1977.

[Avi84] A. Avizienis and P.A. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, pp. 67-80, 1984.

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multiversion Programming IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109,

1986.
[Sco83a] R.K. Scott, "Data Domain Modeling of Fault Tolerant Software Reliability", Ph.D. Dissertation, North Carolina State University, Raleigh, North Carolina, 1983
[Sco83b] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", Proc. IEEE 14th Fault-Tolerant Computing Symposium, pp. 102-107, 1983
[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984
[Sco87] R.K Scott, J. W. Gault and D. F. McAllister, "Fault-Tolerant Reliability Modeling", IEEE Trans. Soft. Eng. Vol. SE-13, No. 5, pp. 582-592, 1987
[Tri82] K.S. Trivedi, "Probability and Statistics with Reliability, Queueing, and Computer Science Applications, Prentice-Hall, New Jersey, 1982.
[Vou85] M.A. Vouk, D.F. McAllister, K.C. Tai, "Identification of correlated failures of fault-tolerant software systems", in Proc. COMPSAC 85, 437-444, 1985.
[Vou86] M.A. Vouk, D.F. McAllister, and K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-tolerant Software", Proc. Workshop on Software Testing, Banff, Canada, IEEE CS Press, July 1986.

Table 3.1 Correctness factors as a function of version reliability under the assumption of version failure independence for 15 functionally equivalent program versions of equal reliability, p. The output space cardinality is r=2, the boundary reliability is 1/r = 1/2 = 0.5.

c_i

i	p=0.49	p=0.50	p=0.51	p=0.80
2	0.6083	0.5000	0.3917	0.0000
3	0.5891	0.5000	0.4110	0.0000
4	0.5696	0.5000	0.4304	0.0001
5	0.5498	0.5000	0.4502	0.0010
6	0.5300	0.5000	0.4700	0.1154
7	0.5100	0.5000	0.4900	0.2000
8	0.4900	0.5000	0.5100	0.8000
9	0.4700	0.5000	0.5300	0.8846
10	0.4502	0.5000	0.5498	0.9990
11	0.4304	0.5000	0.5696	0.9999
12	0.4110	0.5000	0.5891	1.0000
13	0.3917	0.5000	0.6083	1.0000
14	0.3728	0.5000	0.6272	1.0000
15	0.3543	0.5000	0.6457	1.0000

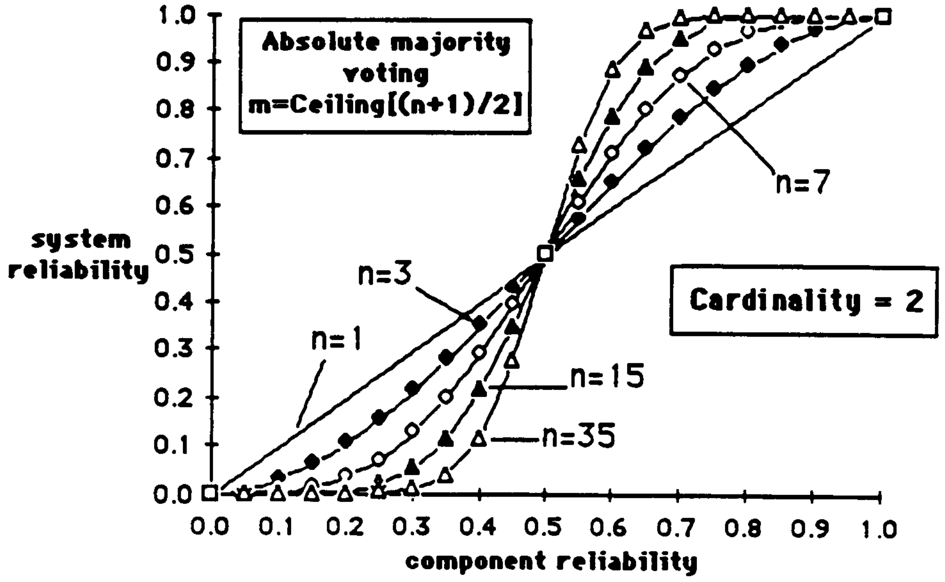


Figure 5.1 System reliability vs. component reliability for absolute majority voting strategy. Number of components used for voting is "n", the agreement number is m, r=2 , and boundary version reliability is 1/r=0.5.

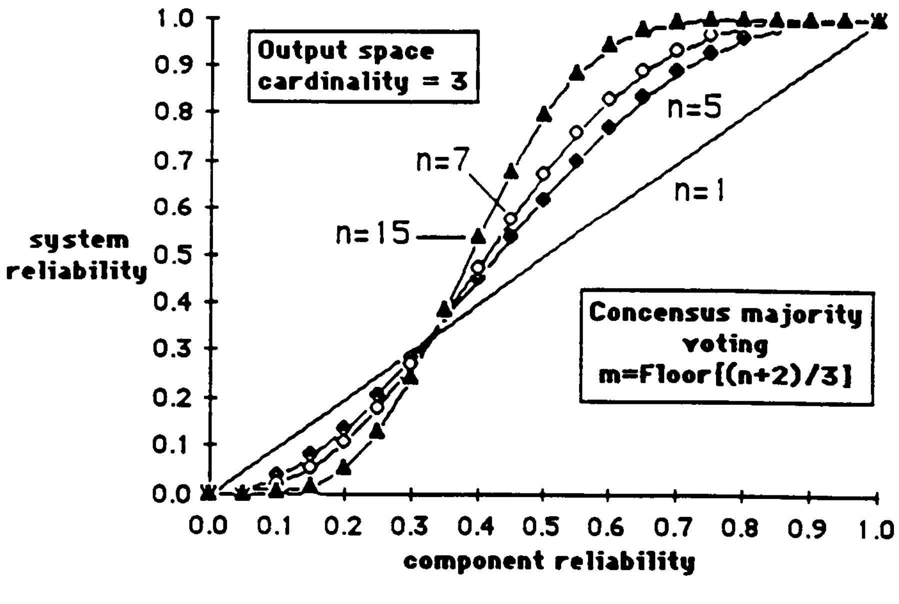


Figure 5.2 System reliability vs. component reliability for the consensus majority voting strategy. The number of voting components is "n", the agreement number is m, r=3, and the boundary version reliability is 1/r = 0.3333.

Table 5.1 The reliability of the N-version programming system using majority voting (r=2). Reliability of the system is p, the number of components participating in a vote is n.

p	n=3	n=7	n=15	n=35
0.0500	0.00725	0.00039	0.00000	0.00000
0.1000	0.02800	0.00273	0.00031	0.00000
0.1500	0.06075	0.01210	0.00361	0.00000
0.2000	0.10400	0.03334	0.00424	0.00003
0.2500	0.15625	0.07056	0.01730	0.00070
0.3000	0.21600	0.12604	0.05001	0.00642
0.3500	0.25175	0.19985	0.11323	0.03363
0.4000	0.35200	0.28979	0.21310	0.11431
0.4500	0.42525	0.39171	0.34650	0.27514
0.4900	*0.48500*	*0.47813*	*0.46861*	*0.45257*
0.5000	*0.50000*	*0.50000*	*0.50000*	*0.50000*
0.5100	*0.51500*	*0.52187*	*0.53139*	*0.54743*
0.5500	0.57475	0.60829	0.65350	0.72486
0.6000	0.64800	0.71021	0.78690	0.88569
0.6500	0.71825	0.80015	0.88677	0.96637
0.7000	0.78400	0.87396	0.94999	0.99358
0.7500	0.84375	0.92944	0.89270	0.99930
0.8000	0.89600	0.96667	0.99579	0.99997
0.8500	0.93925	0.98790	0.99639	1.00000
0.9000	0.97200	0.99727	0.99969	1.00000
0.9500	0.99275	0.99961	1.00000	1.00000
0.9900	0.99990	1.00000	1.00000	1.00000
0.9990	1.00000	1.00000	1.00000	1.00000

Table 5.2 The reliability of the N-version programming system using consensus majority voting (r=3). Reliability is p, the number of components participating in a vote is n.

p	n=3	n=7	n=11	n=15
0.0500	0.00247	0.00131	0.00049	0.00008
0.1000	0.03590	0.01708	0.00646	0.00243
0.1500	0.07843	0.05064	0.02888	0.01584
0.2000	0.13472	0.10502	0.07584	0.05420
0.2500	0.20239	0.17870	0.15164	0.12884
0.3000	0.27884	0.26785	0.25339	0.24154
0.3300	*0.32779*	*0.32662*	*0.32511*	*0.29537*
0.3330	*0.33266*	*0.33258*	*0.33251*	*0.33237*
0.3333	*0.33333*	*0.33333*	*0.33333*	*0.33333*
0.3340	*0.33468*	*0.33484*	*0.33499*	*0.33527*
0.3400	*0.34448*	*0.34683*	*0.34994*	*0.35280*
0.3500	0.36727	0.37141	0.37519	0.38248
0.4000	0.44704	0.47123	0.50430	0.53410
0.4500	0.53321	0.57412	0.62970	0.67710
0.5000	0.61719	0.67090	0.74145	0.79646
0.5500	0.69650	0.75753	0.83292	0.88482
0.6000	0.76896	0.83117	0.90141	0.94252
0.6500	0.83276	0.89030	0.94790	0.97534
0.7000	0.88653	0.93474	0.97604	0.99125
0.7500	0.92944	0.96549	0.99804	0.99758
0.8000	0.96128	0.98458	0.99730	0.99953
0.8500	0.98253	0.99470	0.99947	0.99995
0.9000	0.99448	0.99887	0.99995	0.99999
0.9500	0.99926	0.99992	1.00000	1.00000
0.9900	0.99999	1.00000	1.00000	1.00000
0.9990	1.00000	1.00000	1.00000	1.00000

Table 5.3 The reliability of the N-version programming system using 2-of-n voting strategy (r=∞). System reliability is p, the number of components participating in a vote is n.

p	n=3	n=7	n=11	n=15
0.0500	0.00725	0.00019	0.00001	0.00000
0.1000	0.02800	0.14969	0.30264	0.45096
0.1500	0.06075	0.28342	0.50781	0.68141
0.2000	0.10400	0.42328	0.67788	0.83287
0.2500	0.15625	0.55505	0.80290	0.91982
0.3000	0.21600	0.60758	0.88701	0.96473
0.3500	0.28175	0.76620	0.93492	0.98582
0.4000	0.35200	0.84137	0.96977	0.99783
0.4500	0.42525	0.89758	0.98601	0.99831
0.5000	0.50000	0.93750	0.99414	0.99951
0.5500	0.57475	0.96429	0.99779	0.99989
0.6000	0.64800	0.98116	0.99927	0.99997
0.6500	0.71825	0.99099	0.99980	1.00000
0.7000	0.78400	0.99621	0.99995	1.00000
0.7500	0.84375	0.99865	0.99999	1.00000
0.8000	0.89600	0.99963	1.00000	1.00000
0.8500	0.93925	0.99993	1.00000	1.00000
0.9000	0.97200	0.99999	1.00000	1.00000
0.9500	0.99275	1.00000	1.00000	1.00000
0.9900	0.99970	1.00000	1.00000	1.00000
0.9990	1.00000	1.00000	1.00000	1.00000

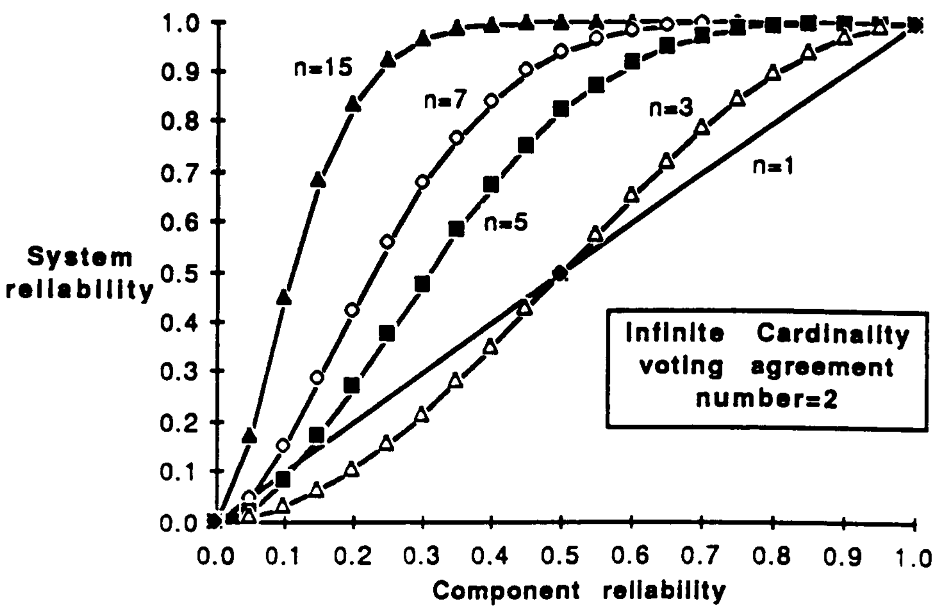


Figure 5.3 System reliability vs. component reliability assuming infinite cardinality of the output space under 2-of-n voting strategy.

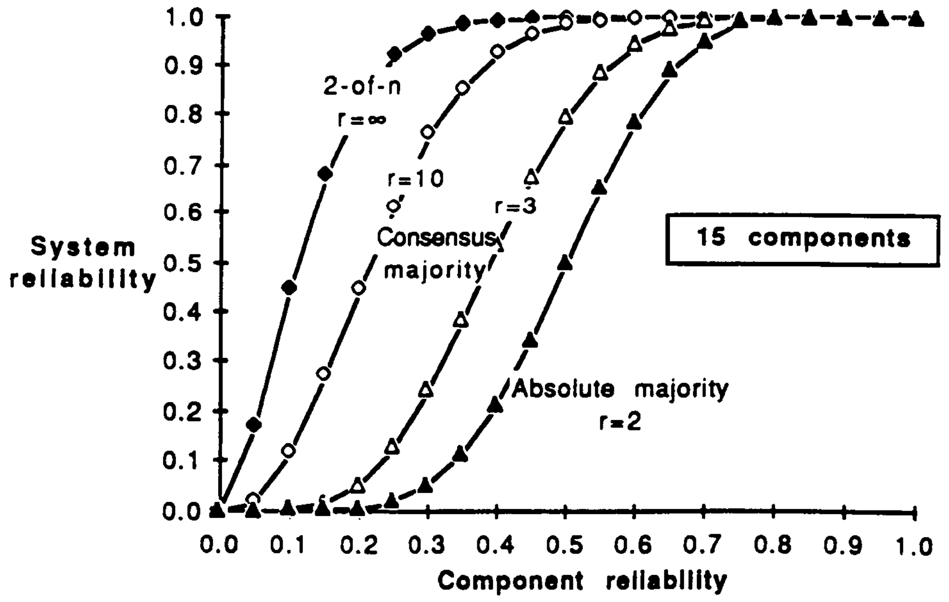


Figure 5.4 System reliability vs. component reliability for n=15 in the range r=2 to r=∞, under appropriate voting strategies. Probability of each j=2..r failure state is (1-p)/(r-1). Simulation was used to compute the r=10 curve.

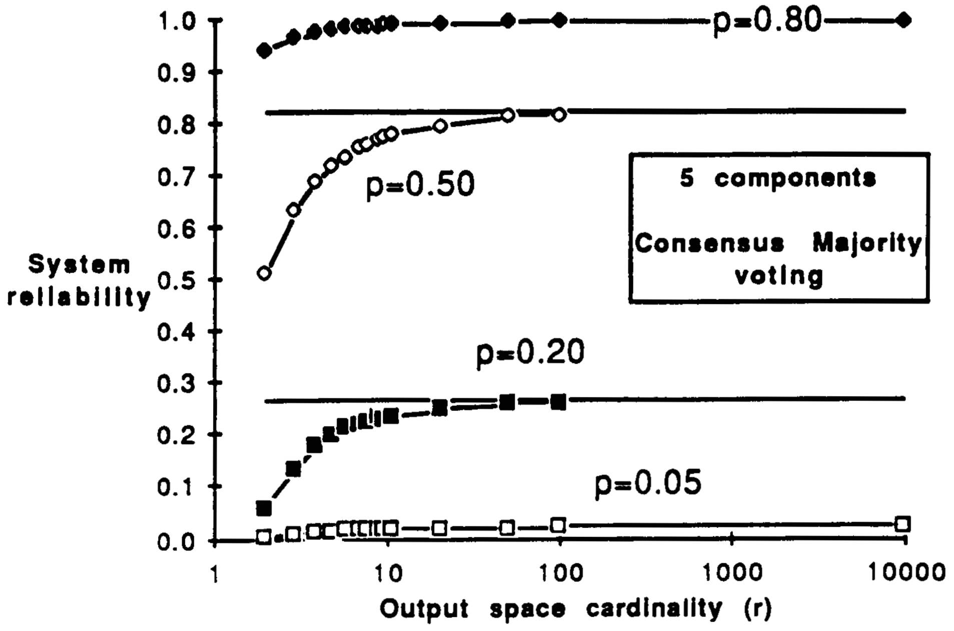


Figure 5.5a System reliability vs. Output space cardinality for n=5 using consensus majority voting. All components have the same reliability, p. Probability of each j=2..r failure state is (1-p)/(r-1). Majority of the data points were computed by simulation.

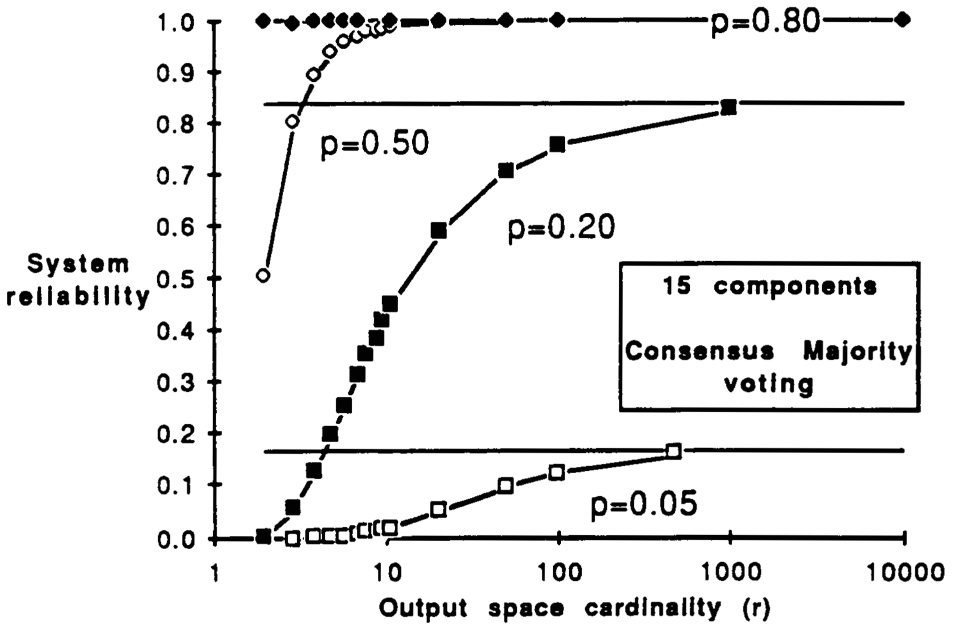


Figure 5.5b System reliability vs. output space cardinality for n=15 using consensus majority voting. All components have the same reliability, p. Probability of each j=2..r failure state is (1-p)/(r-1). Majority of the data points were computed by simulation.

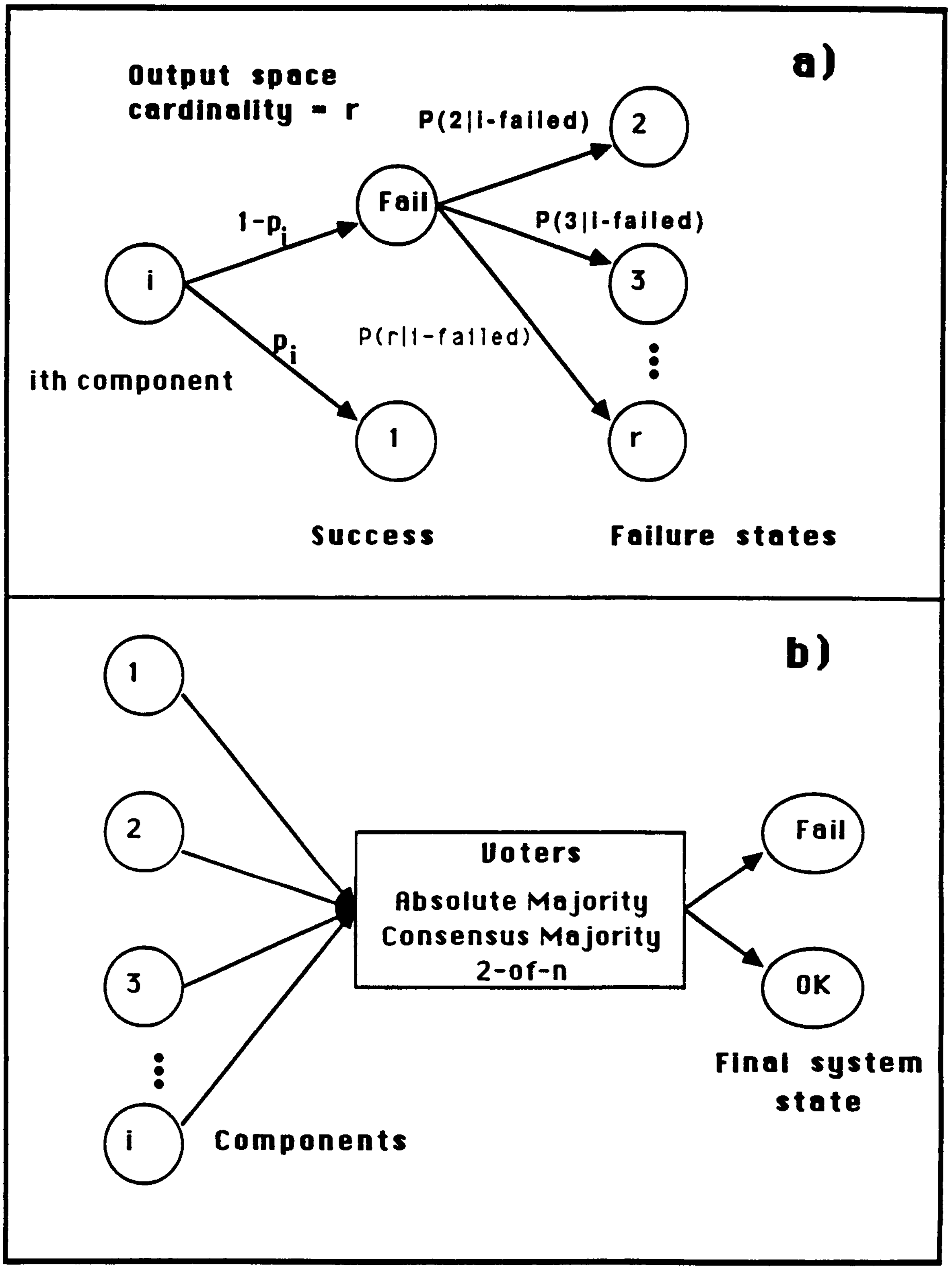
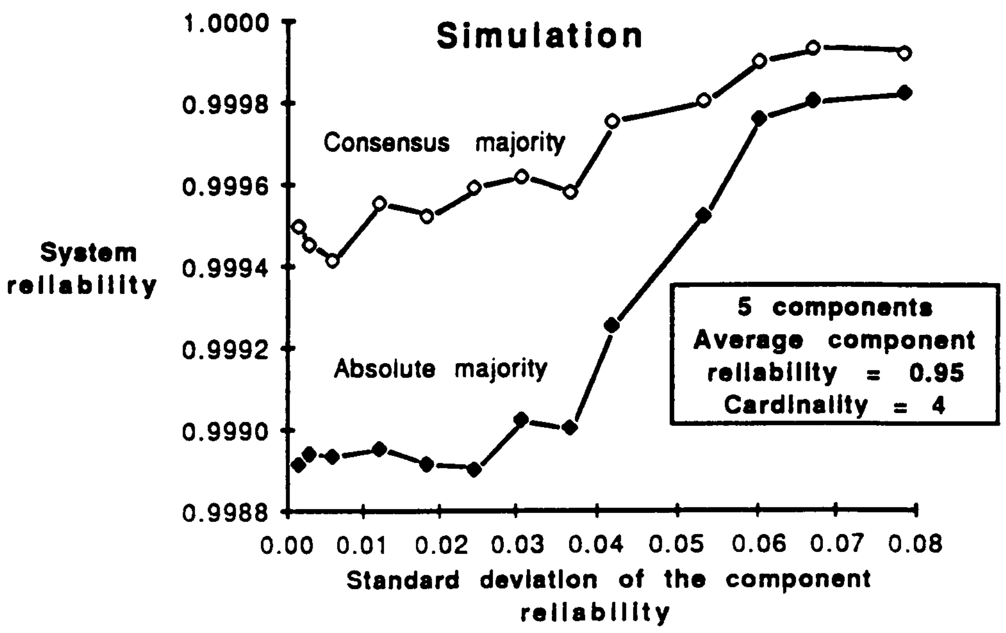


Figure 6.1 Schematic representation of the simulation states for a single component (a), and the voting process (b). Individual component reliability is represented by p_i, and the conditional probability of failing with state j = 2 .. r by P(j|i-failed).



`**Figure 6.2** System reliability vs. the standard deviation of the component reliability. Probability of each failure state is (1-p)/(r-1) = 0.05/3.

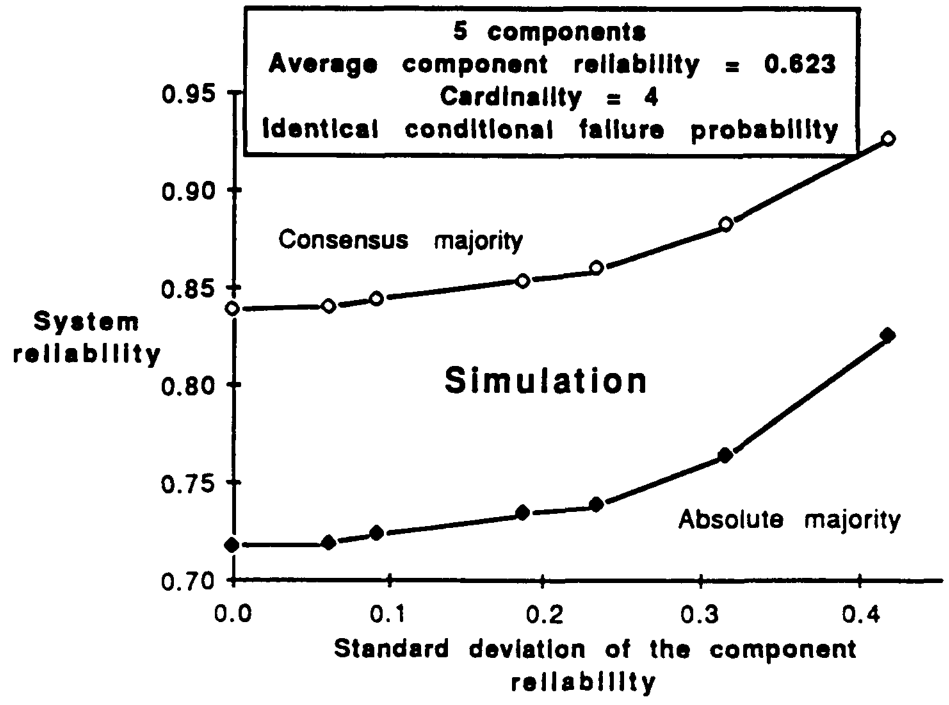


Figure 6.3 System reliability vs. the standard deviation of the component reliability. Probability of each failure state is (1-p)/(r-1) = 0.377/3.

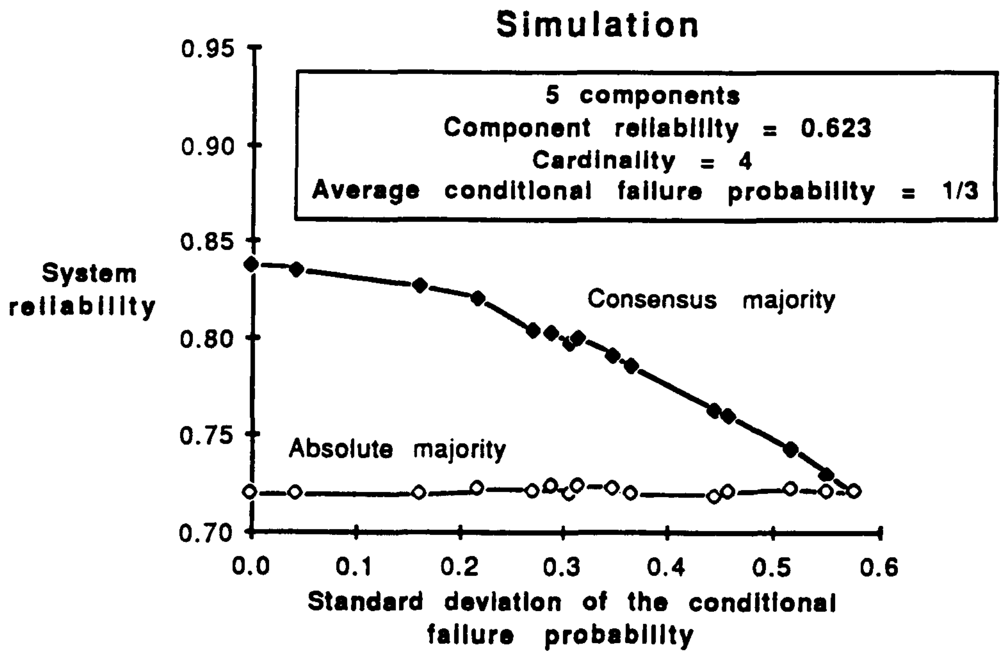


Figure 6.4 System reliability vs. the standard deviation of the conditional failure state probability. All components have identical reliability p = 0.623.

N88-13865

Appendix II

COMPUTER STUDIES

TECHNICAL REPORT

EFFECTIVENESS OF BACK-TO-BACK TESTING

Mladen A. Vouk, David F. McAllister
David E. Eckhardt
Alper Caglayan
John P. J. Kelly

TR-87-08

North Carolina State University

Raleigh, N. C. 27650

EFFECTIVENESS OF BACK-TO-BACK TESTING*

Mladen A. Vouk, David F. McAllister
North Carolina State University
Department of Computer Science, Box 8206, Raleigh, NC 27695-8206

David E. Eckhardt
National Aeronautics and Space Administration
Langley Research Center, Hampton, Va 23665

Alper Caglayan
Charles River Analytics
55 Wheeler St., Cambridge, Ma 02138

John P. J. Kelly
University of California, Santa Barbara
Department of Electrical and Computer Engineering, Santa Barbara, Ca 93106

Key Words: Software fault-tolerance, software testing, back-to-back testing, correlated errors, software reliability

Abstract

Three models of back-to-back testing process are described. Two models treat the case where there is no inter-component failure dependence. The third model describes the more realistic case where there is correlation among the failure probabilities of the functionally equivalent components. The theory indicates that back-to-back testing can, under right conditions, provide a considerable gain in software reliability. The models are used to analyse the data obtained in a fault-tolerant software experiment. It is shown that the expected gain is indeed achieved, and exceeded, provided the inter-component failure dependence is sufficiently small. However, even with relatively high correlation the use of several functionally equivalent components coupled with the back-to-back testing may provide a considerable reliability gain. Implications of this finding are that the multiversion software development is a feasible and cost-effective approach to providing highly reliabile software components intended for fault-tolerant software systems, on condition that special attention is directed at early detection and elimination of correlated faults.

(*) This research was supported in part by NASA grants NAG-1-667, and NAG-1-512, and contract NAS-1-17705.

1. Introduction

Fault-tolerance is or will become part of many critical software and hardware systems [e.g., Mar82, Mad84, Tro85, Bis86]. There are two common methods for achieving software fault-tolerance. These are the N-version programming approach and the recovery-block approach [Ran75, Avi84].

Although existing fault-tolerant software (FTS) techniques can achieve an improvement in reliability over non-fault-tolerant software, experiments show that failure dependence among FTS system components may not be negligible in the context of current software development and testing techniques [Nag82, Sco84, Nag84, Vou85, Wig84, Kni86, Kel86]. Correlated coincidental component failures may be disastrous in current FTS approaches and can seriously undermine any reliability gains offered by the fault-tolerance mechanisms [e.g. Sco83a, Sco84, Avi84, Eck85, Vou86a]. Hence it is important to detect and eliminate them as early as possible in a FTS life-cycle.

Throughout this paper we shall use the terms "component(s)", "version(s)", "functionally equivalent software components", and "software components" interchangeably. The terms "coincident", "correlated" and "dependent" failures (faults) have the following meaning. When two or more functionally equivalent software components fail on the <u>same</u> input case we say that a <u>coincident</u> failure has occurred, and k failing components give a <u>level-k</u> coincident failure. The fault(s) causing a level-k failure we shall call level-k fault(s). When two or more versions give the same incorrect response, to a given tolerance, we say that an <u>identical-and-wrong</u> (IAW) answer was obtained. If the measured probability of the coincident failures is significantly different from what would be expected by random chance on the basis of the measured failure probabilities of the participating components [e.g. Eck85, Kni86, Vou85], then we say that the observed coincident failures are <u>correlated</u> or <u>dependent</u>, i.e. if Pr denotes probability, then

$$\Pr\{\text{ version(i) fails } | \text{ version(j) fails } \} \neq \Pr\{\text{ version(i) fails }\}.$$

If a fault, or a fault combination, results in a IAW answer from k components we say that the falut(s) has (have) "span" of size k. The fault span is important because with the probability of excitation of the fault (fault intensity or visibility) it determines the level of the inter-component failure correlation for that fault.

The back-to-back testing technique discussed here involves pairwise comparison of all functionally equivalent components. Whenever a difference is observed among responses, the problem is thoroughly investigated and appropriate action is taken. If all answers are identical to within a specified tolerance then a "no detected failure" event is said to occur. We say that back-to-back testing fails when all the components fail with (within tolerance) IAW answers. Our experiments indicate that these corrrelated faults occur in practice but can be prevented or reduced.

In section 2 we present three models of the back-to-back testing process. In section 3 we use the models to analyse and discuss the experimental information concerning the effectiveness of the back-to-back testing and the multiversion development approach.

2. Failure Models

Our goal is to model the probability that k versions obtain IAW answers and compare it with experimental findings. We will present three models, two of which assume that the versions fail independently and the third attempts to capture the correlation between versions. We assume that m functionally equivalent software components of versions were developed by independent programming teams from equivalent specifications. We will select a subset of size k from these m versions which we will call a k-tuple.

Back-to-back testing of k components fails to signal a potential error when all the components agree, to a given tolerance, on a value which is a wrong answer. This results in an "undetected" failure. Of course, if k=1 i.e. single component, then every test case (run time errors resulting in operating system intervention excepted) is a potential "undetected" failure in the absence of an oracle or "golden" program. In the following text the term "agreement" means equality between two responses (answers) within a tolerance TOL. It is also assumed that a "golden" or oracle answer is available.

Consider a k-tuple of components. The following two events are independent of the golden program (see examples in Figure 1).

* If all k components agree on an answer, a "COLLECTIVE AGREEMENT" event occurs.
* If there is any disagreement among the components (components being compared pairwise with each other, C(k,2) comparisons in all), a "COLLECTIVE WARNING" event occurs.

The following event depends on the golden answer:

* If all k components agree with the "golden" answer a "SUCCESS" event occurs.

The following three events are called FAILURE events and they also depend on the golden answer:

* If one or more of the component answers disagree with the golden answer a "ONEPLUS FAILURE" event occurs. This is a pessimistic (conservative) view of the failure recognition process, since one or more failures is considered to fail the k-tuple.

The following events are subevents of the ONEPLUS FAILURE event:

* If the majority of the components disagree with the golden answer then we say that a "MAJORITY FAILURE" event has occurred. The majority of k components is defined as $\lceil (k+1)/2 \rceil$. It is possible to define other intermediate states such as the majority

of components agreeing with the golden answer, two-or-more disagreeing etc.

- If all the components disagree with the golden answer then we say that an "ALL FAILURE" event takes place.

Other measure of "distance", such as the difference between the mean value of the component answers and the golden answer, may coalesce the ONEPLUS through ALL FAILURE events into a single event.

Combinations of the above "elementary" events produce the following mutually exclusive and collectively exhaustive back-to-back testing events (see Figure 1):

* If a SUCCESS occurs together with a COLLECTIVE AGREEMENT then an "OK" event occurs.
* If a SUCCESS occurs together with a COLLECTIVE WARNING then a "FALSE FLAG" event occurs. The back-to-back testing signals an error when one is not present. We note that $|a-b| \leq$ TOL and $|b-c| \leq$ TOL does not imply that $|a-c| \leq$ TOL. Hence, FALSE FLAG events are not inconsistent.
* If a FAILURE occurs together with a COLLECTIVE WARNING we say that a "FLAG" event has occurred. The back-to-back testing correctly detected a potential failure (fault).
* If FAILURE occurs simultaneously with a COLLECTIVE AGREEMENT then we say that a NO_FLAG event occurs. This is the most significant back-to-back testing event. A potential failure exists (the failure is fully confirmed if ALL FAILURE has occured) but was not detected by back-to-back testing.

2.1 IAW Models

Formally, the probability that a given subset of k versions or a k-tuple obtains a IAW answer can be written as a conditional probability as follows. Let A denote the event "k versions obtain identical answers" (COLLECTIVE AGREEMENT),

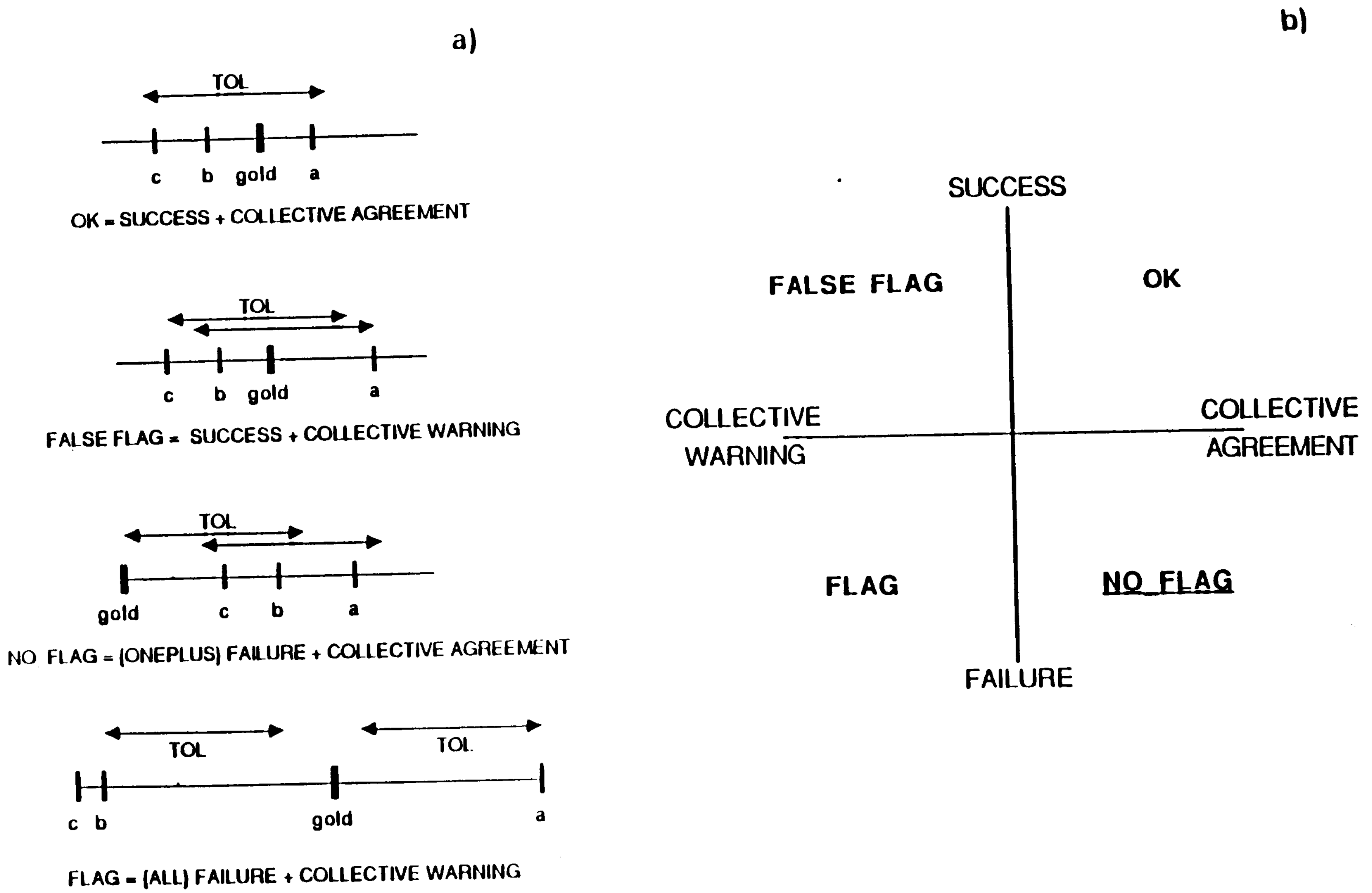

a)
TOL
c b gold a
OK = SUCCESS + COLLECTIVE AGREEMENT
TOL
c b gold a
FALSE FLAG = SUCCESS + COLLECTIVE WARNING
TOL
gold c b a
NO FLAG = (ONEPLUS) FAILURE + COLLECTIVE AGREEMENT
TOL
TOL
c b gold a
FLAG = (ALL) FAILURE + COLLECTIVE WARNING
b)
SUCCESS
FALSE FLAG
OK
COLLECTIVE WARNING
COLLECTIVE AGREEMENT
FLAG
NO FLAG
FAILURE

Figure 1. An illustration of the relationship between the tolerance TOL and elementary back-to-back testing events (a), and of the back-to-back event space (b).

an B the event "k versions fail simultaneously" (ALL FAILURE), and A&B their intersection (NO_FLAG), then

$$P(A\&B) = P(A|B)P(B) \tag{1}$$

Let p_i represent the probability that component i, $1 \leq i \leq m$, fails on a given input, and let $\bar{p}$ be the mean failure probability per test case per component for the set of m components. Then

$$\bar{p} = (\sum p_i)/m \tag{2}$$

where the sum is from i=1 to m.

It is possible to construct C(m,k) sets of k-tuples from a pool of m components, where C(m,k) is the number of combinations of m objects taken k at a time. If the failure probabilities are <u>independent</u> then the probability of an ALL FAILURE for the j th k-tuple is as follows:

$$P_j(k) = \prod p_r = p_1 p_2 \cdots p_k \tag{3}$$

We will use P(k) to denote the average of the $P_j(k)$'s over all C(m,k) subsets:

$$P(k) = [\sum P_j(k)]/C(m,k) \tag{4}$$

where sum is from j=1 to C(m,k). We note that it can be shown that $(\bar{p})^k \geq P(k)$. We also note that if $\bar{p}_k$ denotes the average failure probability of a single k-tuple, then the average of this value over C(m,k) k-tuples is $\bar{p}$. In the following text, unless stated otherwise, all the quantities are averaged over C(m,k) k-tuples.

From the definition of the NO_FLAG event it follows that the probability of this event is less than or equal to the probability of a FAILURE event. Let $P_I(k)$ be the probability of an IAW answer from all k components of a k-tuple (ALL FAILURE event). Then

$$P_I(k) = \gamma(k)Pr(k \text{ versions fail simultaneously}) \tag{5}$$

where $\gamma(k)$ is the conditional probability of an identical level-k answer (given an ALL FAILURE event occurred). This quantit y has , in general, two components. One is due to the cardinality of the output space, and the second component is the failure (fault) dependence. The probability of IAW answers increases as the cardinality of the output space decreases. For low cardinality (finite) output spaces the probability of a coincident failure of two or more components resulting in an IAW answer may be quite high without any correlation being present. For example, if output space is binary, then all programs which are incorrect will produce the same wrong answer, i.e. the probability of IAW answers is 1 for failing versions.

In our first model we approximate the probability of event B in equation (1) by the relationship Pr(k versions fail simultaneously) = $(\bar{p})^k$. Therefore in Model I the probability that back-to-back testing fails to detect an error (NO_FLAG event) is

$$P_I(k) = \gamma_s(k)(\bar{p})^k \qquad (6)$$

where $\gamma_s(k)$ represents the component of $\gamma(k)$ associated with the output space cardinality effect. Since independence is assumed the failure (fault) correlation is zero. The space cardinality component $\gamma_s(k)$ is expected to be a decreasing function of the size of the error output space, x, and the number, k, of the interacting components [Sun85]. Hence, a shape similar to $1/x^k$ would be expected. In practice $\gamma_s(k)$ will also reflect the sampling strategy over the input/output domains, and will be a composite function over all the variables involved in determining the correctness of an answer. See [Sun85] for a more detailed discussion of this phenomenon.

Equations (1), (3) and (4) yield Model II for the failure probability of the back-to-back testing approach:

$$P_I(k) = \gamma_s(k)P(k) \qquad (7)$$

When p_i's are equal to, say p, for all i, then the two models become equivalent, i.e. $P_I(k) = \gamma_s(k)\, p^k$. The difference in the estimates offered by the two models depends on the variance of $\bar{p}$. It can be shown that Model I will always offer a more conservative estimate of the back-to-back failure probability than Model II. Since $\gamma_s(k) \leq 1$, P(k) and $(\bar{p})^k$ provide upper bounds or <u>worst-case</u> values for $P_I(k)$.

As an illustration of the detrimental influence of inter-component failure correlation consider the following. Let $P_c(k)$ denote the average probability of an ALL FAILURE event in an environment where inter-component failure (fault) correlation is present. Then Model III is given by:

$$P_I(k) = \gamma(k) P_c(k) \quad (8)$$

The components of $\gamma(k)$ are

$$\gamma(k) = \gamma_s(k) + \gamma_c(k) - P(s\&c) \quad (9)$$

where γ_c denotes the influence of the fault correlation, and P(s&c) the probability of the intersection of the space and correlation events.

The conditional probability $\gamma_c(k)$ is a function of the number of components containing the fault(s) resulting in an IAW answer (fault span), the visibility (or excitation probability under given sampling conditions, [Ram82]) of the fault(s), and of the number of such faults in the set under consideration.

Let $\gamma_s(k)=0$. This is often a reasonable assumption. Also assume that all the failures are caused by the same fault, or fault combination, and that all failures result in IAW answers from s components, i.e. the fault span is s. Then, given an input which results in failure, and provided $k \leq s$, we can construct C(s,k) k-tuples where all components fail with an IAW answer, and C(m,k) k-tuples in all. Therefore the probability of randomly choosing a k-tuple exhibiting level-k IAW is C(s,k)/C(m,k). If we assume that the probability of failure on input is $\bar{p}$ on the average, then the probability that back-to-back testing fails to signal a level-k failure is

$$P_I(k) = [C(s,k)/C(m,k)]\,\bar{p} \qquad (10)$$

When $s<k$, $C(s,k)$ is defined to be zero. Note that if failures are completely uncorrelated fault span is one ($s=1$), and then $P_I(k>1)=0$

3. Experimental Results

In the summer 1985 a FTS experiment took place sponsored by NASA Langley Research Center. The participants were the authors, the Research Triangle Institute (NC), the University of Illinois (Urbana-Champagne, Il), and the University of Virginia (Charlottesville, Va). A detailed description of the experiment is given in [Kel86]. The programmers worked in two-person teams formed by random selection. All the programmers worked from the same specification. The programming teams were responsible for the software design, the implementation and the testing phases of the life-cycle. The experimenters provided acceptance testing of the product.

The experiment resulted in 20 functionally equivalent programs for solving a problem in inertial navigation. The problem specification was new, written for the experiment, and was not debugged via a "pilot" version of the code prior to the production of the redundant components . This resulted in a very heavy query traffic between the experimenters and the programming teams during the component design phase and in the initial stages of the implementation.

The acceptance testing was a low-expectation process, i.e. only a few critical variables were checked, and only 50 random test cases were used. Consequently the functional and structural test coverage of the products was low. The reliability of the components based on the acceptance testing was about 0.94.

The validation testing using much stricter criteria, a range of tolerances for comparing real number values, test data sets consisting of random and extremal and special value test cases, and providing full functional and linear-block coverage, detected a number of faults of varying prevalence and seriousness. Some of the faults were found to be highly correlated. The reliability of the components was found to be a strong function of the tolerance used for comparisons. Adjudication of the answer correctness was performed using a "golden" or oracle program developed at NCSU. Software development and testing was done on VAX 11/750 and 780 hardware running UNIX 4.2BSD, and MicroVAX II hardware running Ultrix 1.2.

In order to study the influence of component reliability and inter-component failure dependence on the performance of back-to-back testing we have formed subsets of components. The subsets had different average component failure probability ($\bar{p}$), and different inter-component correlation characteristics. Components for the subsets were selected on the basis of their behaviour during different stages of the validation testing. The four subsets which are discussed in this paper are coded 6(2.1), 4(2.1), 9(2.1) and 13(3.1). The first number identifies the number of versions (m) and the second the problem specification update number to which the test data and the golden code used for testing conformed.

3.2 Experimental Measurements

The effectiveness of back-to-back testing was investigated using random test cases. The error detecting power, and the structural and functional coverage provided by the random sets saturated very rapidly. Measured values (e.g. $\hat{\gamma}(k)$) stabilized by the time about 100 cases were run (not an unexpected result, [Vou86a,b]), and hence we used only 200 random test cases. In the back-to-back event space this, of course, expands to 200*C(m,k) event samples and gives acceptable 95% confidence bounds on the back-to-back testing parameters.

To measure the "reliability gain" (or unreliability reduction factor), $G(k)$, offered by back-to-back testing process of k components, as opposed to the development of a single component, we shall use the ratio of the probability of an "undetected" failure in an average single component to the probability of an "undetected" failure in an average k-tuple after back-to-back testing:

$$G(k) = \bar{p}/P_I(k) \qquad (11)$$

Experimentally, single component "undetected" failures were recorded by running functionally equivalent components against a "golden" or oracle program to estimate component failure probabilities. An average value was then computed for the pool of available (operational) components.

The "undetected" multicomponent failures probabilities were computed from pairwise comparisons of responses of all components. The results of the comparisons for each test case were recorded in a (k+1) by k response matrix. The zeroth row of the response matrix (i=0) contains information on the comparison of the components with the golden code. The remaining rows cary information about the mutual comparisons of the components. For example, if the comparison of components i and j detected a difference for a given test case then the entries (i,j) and (j,i) were given value 1, otherwise the value was zero. Unless stated otherwise, a comparison involved eleven variables, or 52 individual values if array elements are counted separately. A difference was signalled if even one of these 52 values differed from the golden value. The results shown in this paper were obtained with TOL=0.0001 absolute for real numbers, and TOL=0 for integers. The response matrices were used to compute the usual multiple component failure profiles [Vou85,86a], intensity profiles [Eck85], and counts of the back-to-back events (see section 2).

Let a hat, ^ , denote experimentally obtained estimates. If $\hat{p}_i$ denotes an estimate of the failure probability of component i (relative to the gold program), then $\hat{\bar{p}}$ was computed by substitution of the $\hat{p}_i$ values into equation (2). Individual $\hat{P}_j(k)$'s were similarly computed by substitution in equation (3), and $\hat{P}(k)$ was then computed using equation (4). The estimate $\hat{P}_c(k)$ was computed from the count of all level-k FAILURE events. The estimate $\hat{P}_I(k)$ was calculated from the ratio [NO_FLAG-count/200*C(m,k)], where the count was over all 200 test cases and over all the k_out_of_m possible k-tuples. The parameter γ was estimated from the ratio [NO_FLAG-count/FAILURE-count]. The analysis was performed for all three FAILURE event categories defined in section 2, i.e. ONEPLUS, MAJORITY and ALL. The results are summarized in Table 1.

To illustrate the relative size of the inter-component correlation among the sets, and order them by correlation level, we compute a function L(k) defined by:

$$L(k) = \hat{P}_I(k)/\hat{P}(k) \tag{12}$$

where L(k) may be regarded as the amplification factor of the worst-case uncorrelated back-to-back testing failure probability required to achieve the observed $\hat{P}_I(k)$. The value of L(k) is always positive and may be larger than 1. Since the output space cardinality is the same for all subsets any differences in the L(k)'s stem from the inter-component failure dependence and therefore can be used to estimate its relative magnitude.

3.3 The Gain

The experimental gain estimates (using ONEPLUS FAILURE events) are shown in Figure 2. Note that the ordinate uses logarithmic scale. The notation used in the legend of this and other figures has the following meaning. The first two letters describe the function that is being plotted. If the first letter is T then the curve is the result of theoretical computations, if it is E the data was obtained

Table 1.

Experimental Results

k	$\hat{G}(k)$	$\hat{p}_I(k)$	$\hat{p}_C(k)$	$\hat{P}(k)$	$\hat{\gamma}(k)$	sample size
Set:	6(2.1)	$\bar{p}$ = 0.379 using ONEPLUS FAILURE events				
2	2.80	0.136	0.615	0.126	0.221	3000
3	9.85	0.0385	0.769	0.0351	0.0501	4000
4	51.7	7.33e-3	0.873	7.85e-3	8.40e-3	3000
5	455.0	8.33e-4	0.947	1.38e-3	8.80e-4	1200
6	inf	0	1.0	2.01e-4	0	200
Set:	6(2.1)	$\bar{p}$ = 0.379 using MAJORITY FAILURE events				
2	12.5	0.0303	0.144	0.126	0.211	3000
3	24.1	0.0156	0.306	0.0351	0.0515	4000
4	284.3	1.33e-3	0.156	7.85e-3	8.55e-3	3000
5	inf	0	0.270	1.38e-3	0	1200
6	inf	0	0.160	2.02e-4	0	200
Set:	6(2.1)	$\bar{p}$ = 0.379 using ALL FAILURE events				
2	12.5	0.0303	0.144	0.126	0.211	3000
3	137.8	2.75e-3	0.0625	0.0351	0.0440	4000
4	1137.1	3.33e-4	0.0313	7.85e-3	0.0106	3000
5	inf	0	0.0192	1.38e-3	0	1200
6	inf	0	0.0150	2.02e-4	0	200
Set:	4(2.1)	$\bar{p}$ = 0.185 using ONEPLUS FAILURE events				
2	1.12	0.166	0.302	0.0276	0.550	1200
3	1.66	0.111	0.376	3.31e-3	0.296	800
4	3.08	0.060	0.420	3.44e-4	0.143	200
Set:	4(2.1)	$\bar{p}$ = 0.185 using MAJORITY FAILURE events				
2	4.19	0.0442	0.0683	0.0276	0.646	1200
3	3.61	0.0513	0.153	3.31e-3	0.336	800
4	18.5	0.010	0.060	3.44e-4	0.167	200
Set:	4(2.1)	$\bar{p}$ = 0.185 using ALL FAILURE events				
2	4.19	0.0442	0.0683	0.0276	0.646	1200
3	29.6	6.25e-3	0.0262	3.31e-3	0.238	800
4	inf	0	0.0150	3.44e-4	0	200

Table 1. (continued)

k	$\hat{G}(k)$	$\hat{P}_I(k)$	$\hat{P}_C(k)$	$\hat{P}(k)$	$\hat{\gamma}(k)$	sample size
Set:	9(2.1)	$\bar{p}$ = 0.366 using ONEPLUS FAILURE events				
2	2.11	0.174	0.562	0.126	0.309	7200
3	4.40	0.0832	0.683	0.0406	0.122	16800
4	9.74	0.0376	0.767	0.0122	0.0490	25200
5	23.4	0.0157	0.830	3.41e-3	0.0189	25200
6	60.9	6.01e-3	0.882	8.71e-4	6.82e-3	16800
7	175.7	2.08e-3	0.926	2.00e-4	2.25e-3	7200
8	658.6	5.56e-4	0.964	4.05e-5	5.76e-4	1800
9	inf	0	1.000	7.25e-6	0	200
Set:	9(2.1)	$\bar{p}$ = 0.366 using ALL FAILURE events				
2	6.72	0.054	0.169	0.126	0.322	7200
3	26.6	0.0137	0.0936	0.0406	0.147	16800
4	93.2	3.93e-3	0.0556	0.0122	0.0706	25200
5	279.5	1.31e-3	0.0347	3.41e-3	0.0377	25200
6	768.6	4.76e-4	0.0226	8.71e-4	0.0211	16800
7	2635.2	1.39e-4	0.0155	2.00e-4	8.93e-3	7200
8	inf	0	0.0117	4.05e-5	0	1800
9	inf	0	0.0100	7.25e-6	0	200
Set:	13(3.1)	$\bar{p}$ = 0.443 using ONEPLUS FAILURE events				
2	2.45	0.181	0.631	0.189	0.286	15600
3	6.53	0.0656	0.731	0.0782	0.0697	57200
4	18.8	0.0232	0.789	0.0311	0.0294	143000
5	56.0	7.76e-3	0.821	0.0119	9.46e-3	257400
6	187.1	2.32e-3	0.839	4.38e-3	2.77e-3	343200
7	739.1	5.86e-4	0.848	1.55e-3	6.94e-4	343200
8	3732.3	1.17e-4	0.853	5.27e-4	1.37e-4	275400
9	31102	1.40e-5	0.854	1.73e-4	1.64e-5	143000
10	inf	0	0.855	5.46e-5	0	57200
11	inf	0		1.67e-5	0	15600
12	inf	0		4.94e-6	0	2600
13	inf	0		1.42e-6	0	200
Set:	13(3.1)	$\bar{p}$ = 0.443 using ALL FAILURE events				
2	4.82	0.0920	0.254	0.189	0.362	15600
3	22.2	0.0199	0.167	0.0782	0.119	57200
4	86.3	5.13e-3	0.123	0.0311	0.0418	143000
5	304.1	1.46e-3	0.0976	0.0119	0.0149	257400
6	1187.8	3.73e-4	0.0810	4.38e-3	4.61e-3	343200
7	6081.5	7.28e-5	0.0686	1.55e-3	1.06e-3	343200
8	57014	7.77e-6	0.0587	5.27e-4	1.32e-4	275400
9	inf	0	0.0501	1.73e-4	0	14300
10	inf	0	0.0422	5.46e-5	0	57200
11	inf	0	0.0347	1.67e-5	0	15600
12	inf	0	0.0273	4.94e-6	0	2600
13	inf	0	0.0200	1.42e-6	0	200

experimentally. The second letter has the following meanings: G = G(k) or gain, L = L(k), c = $\gamma(k)$. The number following the first two letters denotes the number of versions involved in the comparisons (m), and is used to identify the component subset used. If the data are experimental this number may be followed by another letter. Letter A denotes that ALL FAILURE events were used to derive the plotted values, letter M that the MAJORITY FAILURE events were used, and if there is no letter ONEPLUS events were used. For theoretical curves the number of components is followed, in parentheses, by a roman numeral (I, II or III) identifying the theoretical model bound used to compute the values. In the case of Model III the identifier is followed by the span value used in computations.

It is obvious that even in the worst observed case (subset 4(2.1)) the multiversion development coupled with back-to-back testing offers some gain in reliability over the single component development approach. The size of the fault correlation level, as measured by L(k), is illustrated in Figure 3. Experimental $\hat{\gamma}(k)$ estimates are shown in figure 4. The largest inter-component fault-correlation is exhibited by set 4(2.1) and the smallest by set 6(2.1). From Table 1 we see that the most unreliable set is 13 (average failure probability is 0.443), and the most reliable subset is 4(2.1) with an average failure probability of 0.185. The component sets 6, 13 and 9 reach infinite gain (no "undetected" failures (faults)) for 6, 10 and 9 developed components respectively. Using the conservative ONEPLUS FAILURE events, subset 4(2.1) never detects all the potential failures.

The slopes of the curves in Figure 2, and the gain they imply vary among the subsets. The reason for this difference is primarily the inter-component failure correlation. The influence of the average component failure probability of a set appears to be far less important than the correlation effect. For example, the sets 6 and 9 are approximately equally reliable but the lower correlation set 6(2.1) offers

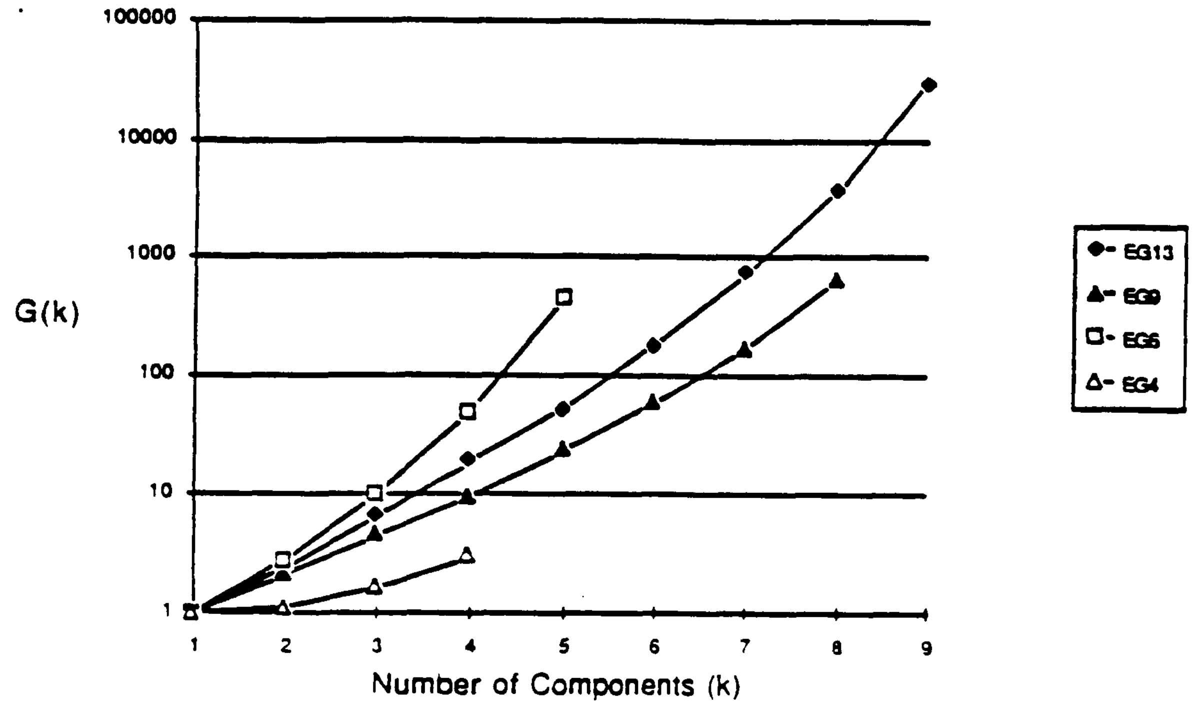


Figure 2. Gain, G(k), vs. Number of Developed components (k). The gain estimate of the ratio of "undetected" failures in an average single component to "undetected" failures remaining after back-to-back testing of the components computed using ONEPLUS FAILURE events.

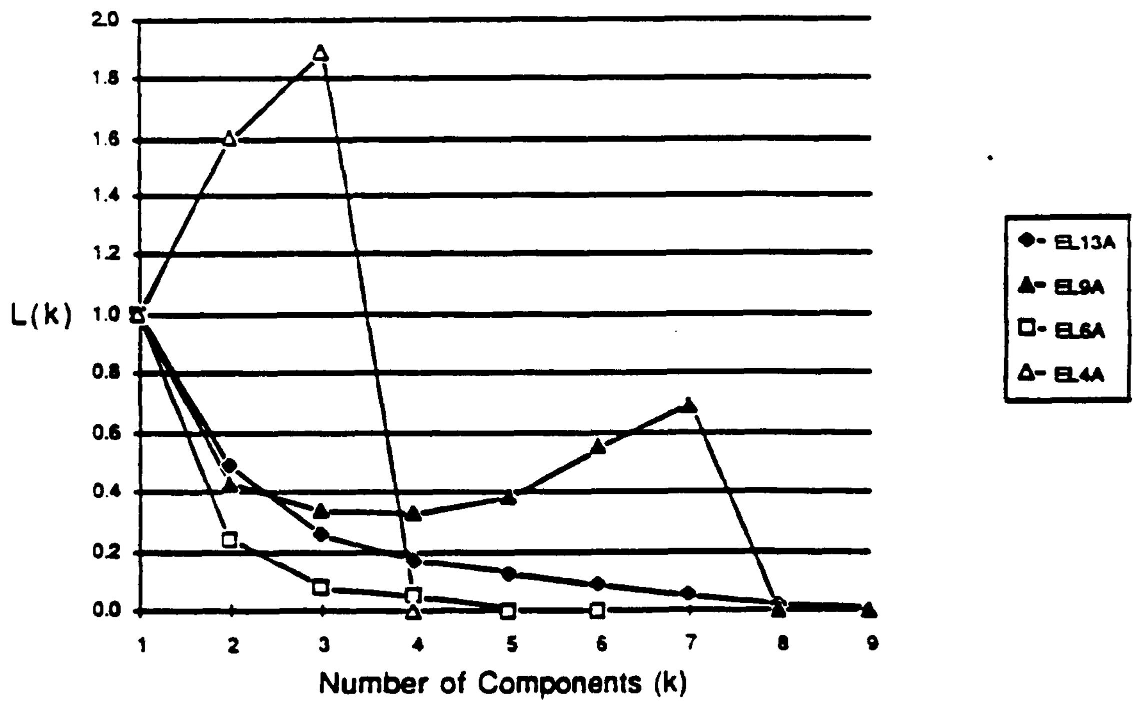


Figure 3. L(k) vs. Number of components(k). Illustration of the relative inter-component correlation. The difference between the curves indicates the difference in the failure (fault) dependence.

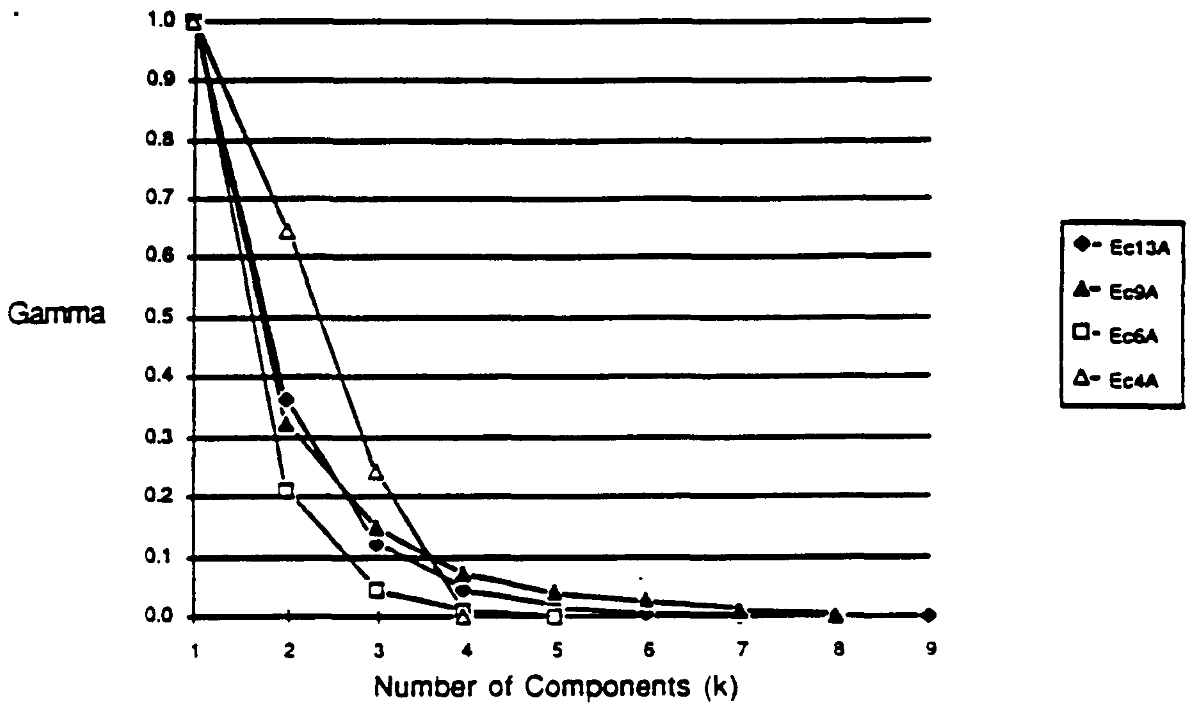


Figure 4. $\gamma(k)$ vs. Number of Components. Experimental estimate $\hat{\gamma}(k)$ computed using ALL FAILURE events.

better gain figures. Similarly, the most "unreliable" set, 13(2.1), has a correlation level which appears to be smaller than that of set 9(2.1), and its gain curve lies above that for the set 9. On the other hand, set 4(2.1) has relatively high reliability, but its components are highly correlated resulting in a gain curve far below any of the other sets.

Figures 5 to 8 show the experimental and theoretical gain curves for each of the component subsets separately. Filled (black) symbols refer to the experimental data and unfilled symbols to theoretical computations. Theoretical computations represent worst-case bounds obtained using Model I (triangles), Model II (squares), and Model III (diamonds, equation (10) using the maximum fault span observed for conservative gain estimates). In the case of set 4(2.1) this last limit would be constant and equal to one, so diamonds in that case represent computations for the span of 3 recorded using the ALL FAILURE events.

It should be noted that the theoretical models, as defined in section 2, do not account for the tolerance effect (i.e. a range of FAILURE events from ONEPLUS to ALL), but only for the ALL FAILURE events. Therefore the theoretical values obtained using these models will underestimate the actual failure probability as measured by FAILURE or MAJORITY FAILURE events. Hence, to validate the models we use the ALL FAILURE event data. Also note that any result checking during the development/testing of components effectively acts as an additional version (even manual computations may qualify as a "version"). Therefore, in practice the minimal number of "developed" components is usually 2.

Figure 5 shows the FAILURE (EG6), MAJORITY FAILURE (EG6M), and ALL FAILURE (EG6A), estimates of the gain for the six component set. Also shown are the worst-case gain curves expected using Model I, TG6(I), and Model II, TG6(II), as well as a Model III based bound (equation (10) with $m=6$, $s=5$, $\bar{p}=0.379$), TG6(III/5). It is interesting to observe that for the ALL FAILURE events the maximum fault span is one less than it is for the ONEPLUS FAILURE events. The conservative experimental gain curve is well approximated by the Model II worst-case bound, while the MAJORITY and ALL FAILURE estimates are better then this bound.

Figure 6 shows the gain curves for the subset 9(2.1). Only the ONEPLUS FAILURE and the ALL FAILURE experimental data are given. The component sets 6(2.1) and 9(2.1) have very similar average component failure probabilities, but they have significantly different inter-component failure dependence characteristics (see Figure 3). The effect of the increased inter-component failure correlation in subset 9(2.1) manifests as a reduced slope of the 9 gain curves. The conservative gain

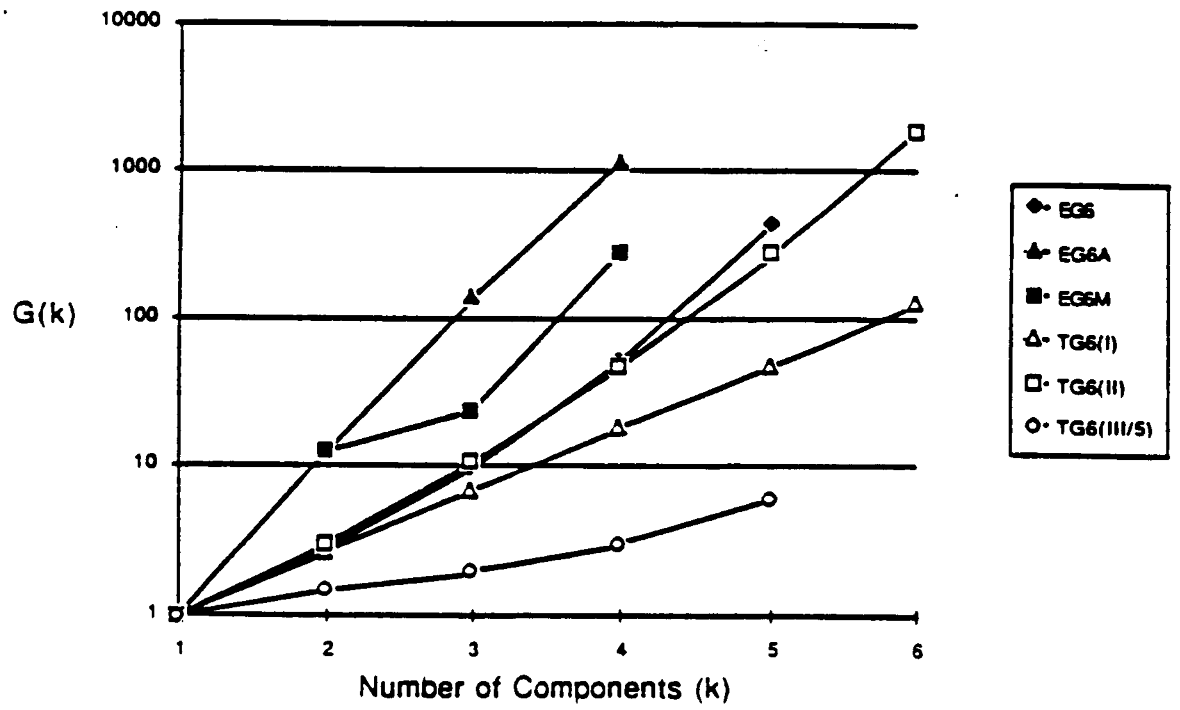


Figure 5. G(k) vs. Number of Components. Experimental and theoretical gain curves for set 6(2.1).

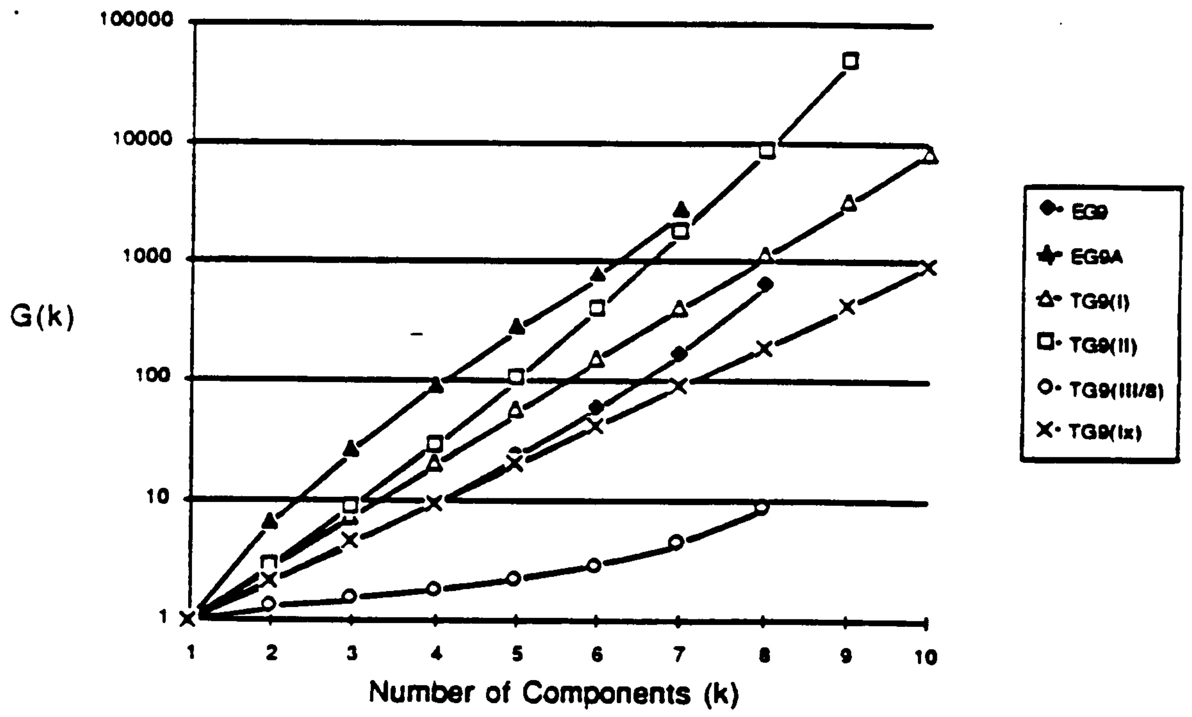


Figure 6. G(k) vs. Number of Components. Experimental and theoretical gain curves for set 9(2.1).

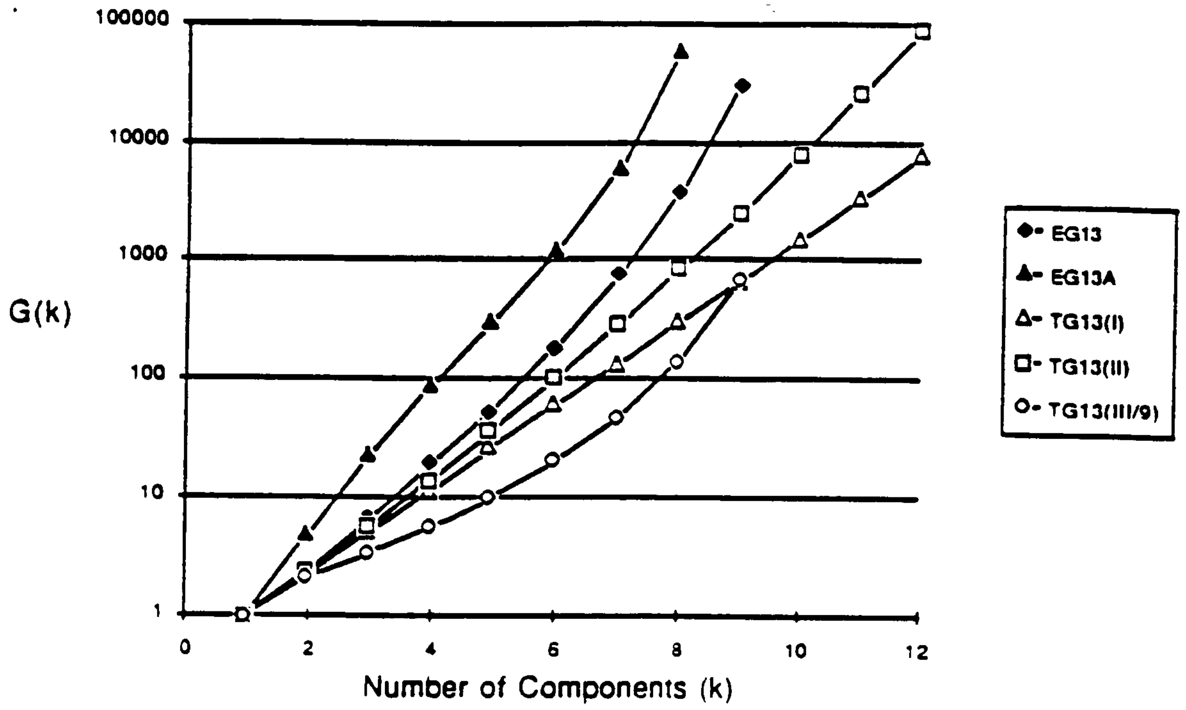


Figure 7. G(k) vs. Number of Components. Experimental and theoretical gain curves for set 13(3.1).

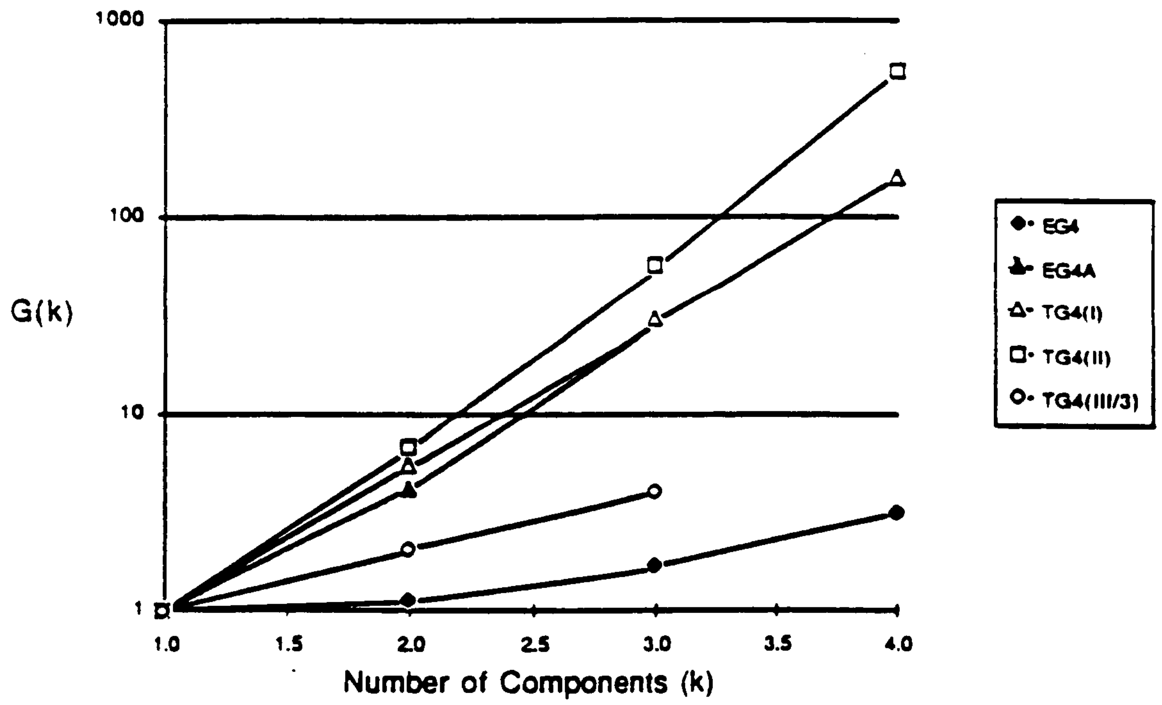


Figure 8. G(k) vs. Number of Components. Experimental and theoretical gain curves for set 4(2.1).

estimates fall below the Model I predictions based on the average failure probability of the whole subset. Figures 7 and 8 illustrate the gain information for subsets 13(3.1) and 4(2.1) respectively.

Considering all four sets we note that the Model I worst-case bound provides a satisfactory lower limit with respect to all ALL FAILURE experimental curves. A reasonable conservative limit seems to be provided through the Model III bound. Work in progress at NCSU shows that good estimates of the correlation behaviour and of the bounds can be obtained <u>without</u> the use of a special golden program. For example, curve TG9(Ix) was computed using Model I with an estimate of $\bar{p}$ based <u>only</u> on the relative performance of the 9 components. Each component was in turn treated as the gold program and average failure probability of the other components was computed relative to it. A grand average was then computed over all the estimates for us in Model I.

4. Conclusions

Using functionally equivalent software components we have experimentally investigated the effectiveness of back-to-back testing process. We compared the unreliability offered by a multiversion development approach with back-to-back testing, with the average unreliability of a single component. Even conservative estimates indicate a considerable increase in the probability of detecting failures (faults) if back-to-back testing is used. Three models of the back-to-back testing process were presented, and it was shown that they offer good estimates of the lower bounds on the observed multiversion development reliability gains.

5. References

[Avi84] A. Avizienis and J.P. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, pp. 67-80, 1984

[Bis86] P.G. Bishop, D.G. Esp, M. Barnes. P Humphreys, G. Dahl, and J. Lahti, "PODS—A Project on Diverse Software", IEEE Trans. Soft. Eng., Vol. SE-12(9), 929-940, 1986.

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Kel86] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "Early Results from the Second Generation Multi-Version Software Experiment", submitted for publication, 1986.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multiversion Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[Mad84] W.A. Madden, and K.Y. Rone, "Design, Development, Integration: Space Shuttle Primary Flight Software System", Comm. of the ACM, Vol. 27(8), 902-913, 1984.

[Mar82] D.J. Martin, "Dissimilar Software in High Integrity Applications in Flight Controls", Proc. AGARD - CP 330. 36.1-36.13, September 1982.

[Nag82] P.M. Nagel and J.A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling", BSC-40366, Boeing, Seattle. Wa., 1982

[Nag84] P.M. Nagel, F.W. Scholz and J.A. Skrivan, "Software Reliability: Additional Investigation into Modeling with Replicated Experiments", NASA CR172378, Boeing, Seattle, Wa., 1984

[Ram82] C.V. Ramamoorthy and F.B. Bastani, "Software reliability - status and perspectives", IEEE Trans. Soft. Eng., Vol. SE-8, 354-371, 1982

[Ran75] B. Randell, "System structure for software fault-tolerance", IEEE Trans. Soft. Eng., Vol. SE-1, 220-232, 1975

[Sco83,a] R.K. Scott, "Data Domain Modeling of Fault Tolerant Software Reliability", Ph.D. Dissertation, North Carolina State University, Raleigh, North Carolina, 1983

[Sco83,b] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", Proc. IEEE 14th Fault-Tolerant Computing Symposium, pp. 102-107, 1983

[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984

[Sco86,b] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling", IEEE Trans. Software Eng., 1986, to appear

[Sun85] C. Sun, "Reliability of N-version programming for finite output spaces", M.Sc. Thesis, North Carolina State University, Raleigh, North Carolina, 1985

[Tro85] R. Troy and C. Baluteau, "Assessment of Software Quality for the Airbus A310 Automatic Pilot", Proc. FTCS 15, Ann Arbor, USA, (IEEE CS Press), 438-443, June 1985.

[Vou85] M.A. Vouk, D.F. McAllister, K.C. Tai, "Identification of correlated failures of fault-tolerant software systems", in Proc. COMPSAC 85, 437-444, 1985.

[Vou86a] M.A. Vouk, D.F. McAllister, K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-tolerant Software", Proc. Workshop on Software Testing, Banff, Canada, IEEE CS Press, July 1986.

[Vou86b] M.A. Vouk, M.L. Helsabeck, K.C. Tai, and D.F. McAllister, "On Testing of Functionally Equivalent Components of Fault-Tolerant Software", Proc. COMPSAC 86, 414-419, 1986.

[Wig84] J.E. Wiggs, "Experimental Validation of Fault-Tolerant Software Reliability Models", M.Sc. Thesis, North Carolina State University, Raleigh, North Carolina, 1984

Appendix III

NASA/NAG-1-667/R/3.0/NCSU.CSC.(DFM,KCT,MAV)/Oct-87

RSDIMU Acceptance Testing System

Version 3.0

May 1987

Fault-Tolerant Software Experiment

OVERVIEW OF ACCEPTANCE SOFTWARE AND PROCEDURES

RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

This note applies to the releases 3.0 of RSDIMU-ATS. The release complies with the version 3.2/10-Feb-87 of the RSDIMU specifications. This system is released for restricted use by sites involved in the NASA-LaRC fault-tolerant software experiment.

Institutions participating in the program:
NASA-Langley Research Center, CRA, NCSU, UVA, UCLA
(ex-participants: RTI, UIUC).

Re-distribution and use of this system for purposes other than the ones compatible with the current experiment is prohibited unless explicit permission is obtained from the NASA-Langley Research Center coordinator for this experiment (Dr. D.E. Echkardt).

The RSDIMU Acceptance Testing System (RSDIMU-ATS) was built to help test and analyse multiversion RSDIMU procedures generated as part of a NASA sponsored fault-tolerant software experiment in progress since Spring 1985. RSDIMU-ATS is intended for use in a UNIX environment and may need to be modified if UNIX-like, or non-VAX systems are used. Part of the software needs to be recompiled if used on non-VAX hardware (e.g. SUN workstations). It was tested on VAX 11/780,785, and MicroVAX II hardware under UNIX 4.2/4.3BSD, and Ultrix1.1/1.2 respectively.

The system is shipped on a 9-track magnetic tape (standard size) in tar format at 1600 bpi, as directory tree rooted in ./fts87.

```
                           fts87
 __________________________|________________________________________
|        |       |     |        |          |          |          |
ReadMe  accept  code  data generators* gold nonVAX_host** certify testcases*
```

  * shipped on request only
  ** to VAX sites shipped on request only

The following notes discuss more important features of the system.

1. ReadMe is the file containing this note.

   RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

2. The core of the acceptance testing system is located in the directory "accept". It is intended to help an experimenter run, evaluate and request corrections of programs in a semiautomatic fashion. A more detailed description of the accept system is found in ReadMe notes in the accept/ and certify/ directories. The system is based on the use

of a "golden" program for adjudication of the correctness of the answers generated by the code being tested. The RSDIMU code is tested one component (version) at the time against a pre-recorded output expected from the "golden" code. Any differences from the expected answers have to be examined in detail, and corrections justified in error correction reports. Our confidence in the correctness of the "golden" code is very high, however experimenters should still be on the look-out for discrepancies indicating possible "golden" code errors. If such are discovered all on-site testing should be frozen, and the RSDIMU-ATS distribution site (NCSU) notified immediately. Similarly, if errors are discovered in the ATS harness scripts please notify NCSU. A detailed description of the acceptance procedure is located in accept/ ReadMe notes. The basic idea is to follow an iterative correction process, i.e test-correct (one or more errors at the time)-test etc. All communication with programmers should follow guidelines given in certify/ReadMe notes. Testing involves location of the fault(s) causing the first 20 failures and its (their) removal. This repeats until no errors are detected by the supplied acceptance test set.
The same test data set is used on all programs at all sites.

An extended analysis system providing MCF profile (intensity) analysis is available on request, but it should not be used to perform acceptance testing as part of the current experiment (see note 4).

3. All the testcases in this release of the system comply with the specification version 3.2/10-Feb-87. A test case entry consists of an input record, and an output record. The latter contains what is believed to be the correct answer to the input record according to the 3.2 specs, and as generated/given by gold3v2.i.

   A set of test cases was designed and generated for acceptance testing rsdimu code. It consists of a group of 796 extremal/special value (ESV) test cases and a group of 400 random test cases. All test cases are located in the directory data/.

   A successful pass through all the test cases gives an estimated lower limit on the reliability of the rsdimu code of about 0.992 (valid for the employed sampling profiles).

   More details about the test cases can be found in the ReadMe notes in the data/ directory.

   Directory "generators" contains files and code one may need to generate the ESV and random data from scratch. It is not expected that a user working in a VAX-UNIX environment would have to do that (however, see notes in the nonVAX_host directory). In fact, it is not recommended that sites generate (or re-generate) data without consultation since rounding and other subtle differences might appear between the newly generated cases and the ones on this distribution tape, resulting in the use of somewhat different cases by different sites.

   Directory testcases/ contains raw esv test cases (text form).

4. Directory "code" is assumed to contain the rsdimu code that is being tested. If you do not wish to keep your code in that directory either use

pointers/links to the code, or change the appropriate portions of the acceptance system.

Note that RSDIMU-ATS is sensitive to the overall file structure that is used in the system, and any changes should be made only after consultation with NCSU.

It is extremely important for the success of the experiment that you keep not only the final, corrected version of each program, but that each intermediate version submitted for acceptance testing is saved and tagged with an appropriate version number and information about the changes/corrections (using the provided change form, in certify).

It is important that you promptly review all the returned error-reports. In this experiment we are not implementing back-to-back version testing and a formal automatic correlation search-and-remove loop (we have the tools, but since this was not part of the original experimental design we do not want to change the rules now). However, all the discrepancies between individually submitted and tested components and the "golden" answers should be scrutinized with extreme care in case the difference is due to the "golden" code, rather than your own code (a posible cause could be use of RSDIMU-ATS in an environment not 100% compatible with the ones in which the system was tested, see notes in nonVAX_host/ directory).

5. The "certify" directory contains code and files that are sent to each maintenance/certification team. It contains a basic rsdimu driver (to avoid interface problems) and instructions on its use. It also contains a sample input and output, and an electronic error report file.

```
******************************************************************
*   Local site experimenters have to change/adjust the           *
*   certify ReadMe and ReadMe_to_certify notes to reflect local  *
*   electronic mail address, and local version management       *
*   environment and approach (RCS, SCCS, or similar).            *
*                                                                *
*   You should also make all 'fts_' files in certify 'read_only' *
*   to prevent accidental changes in the team's testing          *
*   environment (prevent changes by making yourself the owner.   *
******************************************************************
```

Whenever a change is made in the code it is expected that the programmer(s) will record it using this report. A new version of the code, and error and change report(s), have to be returned to the experimenters together.

It is essential that each program be given a version number and associated with it the date of it creation. Every time a program is corrected its version changes and should be recorded in the correction report, as comments in the code itself, and should reflect in the file name for the new code (as kept in the "code" directory, and in the "newcode" directory in accept/).

All versions of the code, i.e. all submitted corrected code, should be preserved for future analysis. Examples are given in the "certify" directory and in the "accept" (detailed procedure). Use magnetic tapes.

If (disk or tape) space is a problem, we suggest that you store only the difference between the original version and successive corrections using for example Unix diff processor.

FOR ANY INFORMATION REGARDING THIS SYSTEM PLEASE CONTACT:

M.A. Vouk
North Carolina State University
Department of Computer Science, Box 8206
Raleigh, NC 27695-8206

Tel: 919-737-7886 (office)
     919-737-2858 (departmental office)

USENET: mcnc!ece-csc!vouk
ARPA: vouk@ece-csc.ncsu.edu

Fault-Tolerant Software Experiment

ACCEPTANCE ENVIRONMENT

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

This is the basic acceptance environment. Most of the shell scripts and programs have either a help which describes its activation parameters (invoke the script without any parameters), or internal documentation. Where needed program source code is provided.

Current version of the system is intended for UNIX csh environment under either 4.2/4.3BSD, or Ultrix 1.1/1.2, running on VAX hardware.

Comparison of the values computed by programs with those using golden code is done using relative tolerance. It is possible to switch to absolute tolerances if that is desired. Do not do that for acceptance testing.

Testing tolerances are set to the following values within fts_accept and can be changed by modifying statements within fts_accept (see fts_accept help):

```
DiffBestEst = 0.00024414
DiffLinOut  = 0.00024414
DiffOffset  = 0.00024414
tolerance_mode = relative
```

Please do not use different tolerances for your acceptance testing before getting concurrence from all the other testing sites. Otherwise we shall each end up testing and correcting different things.

There is no tolerance regarding the display values (five digits), but one could allow for a difference in the last displayed digit. To avoid display related warnings and "failures" of the type: gold 4.9999 vs. computed 5.0000, and to therefore test only the display algorithm, we inject (using voteestimates) golden values for bestest and other real-valued variables prior to display computations.

The acceptance harness tests for agreement on eleven output variables (infact 59, if elements of arrays are counted separately, number of elements is given in parentheses). They are:

LINOFFSET, LINNOISE, LINOUT, LINFAILOUT, SYSSTATUS, BESTEST, CHANEST, CHANFACE, DISMODE, DISUPPER and DISLOWER.

Critical variables are: (3)BESTEST, (8)LINFAILOUT, ((1)SYSSTATUS)

Non-critical: (1)DISMODE, (3)DISUPPER, (3)DISLOWER, (12)CHANEST, (4)CHANFACE

Intermediate: (SYSSTATUS), (8)LINOFFSET, (8)LINNOISE, (8)LINOUT

All variables are checked for each test case.

For more details on the checking of variables and tolerance used see the listings of the fts_harness files, and the April 86 NCSU Working Notes from the Langley meeting (NASA.FTS/NCSU/WN/1/Apr-86), and UCLA notes from the same meeting.

To avoid accidental correctness problems output variables are "trashed" before each test case is run.

The trash values injected in the various output variables of rsdimu are as follows :

```
LINOFFSET : -9999.0
LINOUT    : 999999.0

BESTEST.ACCELERATION [1..3] : 9999999.0;
CHANEST [1..4].ACCELERATION [1..3] : 9999999.0;

DISMODE : 65534
DISUPPER [1..3] : 65534
DISLOWER [1..3] : 65534
```

The values for real variables (first four listed above) cannot occur for the current set of input data, and are highly unlikely otherwise. The display values are supposed to turn on only the G segment for the least significant digit. Boolean output variables, and user defined are initilized by the compiler (to zero).

Primary scripts:

```
fts_certify       - shell script which activates fts_accept with all.dat
                    test data and produces a correction request report
                    and test cases for the cerification team by
                    running fts_correq. Certain program naming conventions
                    and running options are built-in.

fts_accept        - shell script for constructing, compiling and running
                    harness+rsdimu code.

fts_correq        - correction request generation shell script, generates
                    a report/request suitable for mailing to the
                    maintenance teams.
```

Utility scripts and programs:

```
fts_listdata      - script for listing test cases from the test data files.

fts_prnt          - data listing program.

fts_prterr        - program produces test cases suitable for use by
                    fts_driver.p code.

fts_ter1          - block coverage computation script.

fts_lc            - lower-case filter.
```

```
fts_uc              - upper-case filter.

fts_nc              - comment-delimiters lex-based filter.


Source code and script parts:

fts_io              - sed control code to flag "integer","real",and rsdimu i/o.

fts_dbxbug          - dbx bug control code.

fts_dbxinit         - dbx initialization code.

fts_harness.declare - test harness declarations.

fts_harness.rest      - test harness body.

fts_msgtext         - correction request message.

fts_prnt.p          - source code for fts_prnt.

fts_prterr.p        - source code for fts_prterr


Documentation and examples:

ReadMe               - general information about the "accept" directory.

ReadMe_to_certify - certification procedure using fts_certify

ReadMe_accept        - using fts_accept.

example/            - directory with example outputs from an fts_accept run
                      (ncsuD7.i tested by executing:

                      fts_accept ncsuD7.i ncd7 all -c -x > test.ncd7all&
                      fts_correq ncd7 all > correq.ncd7
                      ).

newcode/            - empty directory for testing results (reminder),


Data links:

all.dat             - symbolic link to esv+random acceptance test cases.

esv.dat             - link to extremal and special value (esv) test cases.

randNCSU.dat        - link to independent random test cases (uniform profile).

randCRA.dat         - link to independent random test cases (shaped profile).

                All .dat files are in ../data.
                It is also assumed that the code to be tested is in ../code.
```

```
The fts_lc, fts_uc, fts_nc, and initial fts_io filters were written by RTI.

  The filter fts_io (integer/real, and i/o) is rather crude.
  It will miss 'real' at the begining of a line.
  It may also cause false warnings regarding use of integer and real types,
  and of i/o in the rsdimu code. In those cases hand editing and
  recompilation of <work_name>.p (rsdimu+harness) files may be
  necessary. If editing, search for two question marks ??.
  If recompiling use: pc -s -C -g -z options. Re-run fts_accept without
  the -c option.

  Note that -s compiler option yields messages regarding non-standard
  use of Pascal in the code (primarily the harness code). These
  messages should be ignored.

  Alternatively, delete the first two lines (real/integer),
  or third and fourth lines (i/o) of the fts_io code.

  fts_nc filter for comments may cause problems by making nested
  comments of type {(* comment *)} transform to {{ comment }}
  which is illegal. This filter can be excluded from the processing
  pipe in fts_accept.
```

Fault-Tolerant Software Experiment

PROCEDURE FOR ACCEPTANCE TESTING

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

```
*************************************************************************
*                                                                       *
*  For the purpose of conducting the acceptance testing (certification *
*  of the 20 programs) it is recommended that you use fts_certify.     *
*  Unless you intend to use fts_accept directly, rather than through   *
*  the fts_certify facility, you do not need to read this file.        *
*                                                                       *
*************************************************************************
```

****

It is assumed that the communication between the experimenters and the maintenance personnel will be via electronic mail. It is further assumed that the experimenter has full access to maintenance personnel files, but the reverse is not true. It is also assumed that the acceptance testing is performed in csh in UNIX 4.2/4.3BSD, or Ultrix 1.1/1.2 environments running on VAX hardware (else see nonVAX_host/ directory).

I.

The testing begins by placing the code which is to be tested into the code/ directory. Enter the accept/ directory and start the initial round of testing by executing the fts_accept shell script. It is recommended that you use the -x option and benefit from the coverage information thus provided. For example:

```
     fts_accept ncsuB2.i ncb2 all -c -x > test.ncb2all&
```

NOTE: YOU CANNOT RUN TWO fts_accept JOBS FROM THE SAME DIRECTORY AT THE SAME TIME (IN BACKGROUND). THE SYSTEM WAS NOT DESIGNED FOR THAT AND YOU CAN END UP WITH A MESS. YOU CAN, HOWEVER, KEEP TWO DIRECTORIES, SAY ACCEPT1 AND ACCEPT2 AND RUN AN fts_accept FROM EACH OF THEM WITHOUT INTERFERENCE.

The run should either result in error messages (exit code <= 8) or should complete successfully (exit code 11). When the background job ends check the test.<name>all file carefully.

* No compiler errors or missing voter call problems (exit status > 5).

* Fatal execution time errors exit status = 7.

* Differences detected from expected output exit status =8.

II.

If the test run ends with any status but exit(11), i.e. complete success, produce an error correction request for the maintenance team by running fts_correq script. For example:

```
    fts_correq ncb2 all > correq.ncb2
```

Make sure that the number of failures you wish to analyse is set to 1.
For this see fts_correq code (run fts_correq without parameters).
File errdata.ncb2 will contain input data for failed cases in a form
that is suitable for use by the "fts_driver.p" code in certify/.
Check the content of correq.ncb2 and mail it to the maintenance team
working on the <name> code (in examples: ncsuB2.i and higher versions,
i.e. <name>=ncb2, or ncb3 etc).

You may also have situations where you need to send non-standard messages
as part of the correction request. For example, people may send you
code and reports with incorrect or inappropriate version numbers.
In situations like that create and insert the message at the begining
of the correq.<name> file, just after the standard initial paragraph.

III.

Now create a subdirectory that will hold the starting, and all subsequent
versions for a particular program(mming team). For example:

```
    mkdir newcode/ncsuB
```

make a sub-subdirectory for the current program version:

```
    mkdir newcode/ncsuB/v2
```

and move all the files you want to keep into that sub-sub directory, e.g.

```
    mv *ncb2* newcode/ncsuB/v2
```

You may wish to use diff and compress processors to reduce stored
file sizes.

Unless you are interested in doing further correlation analysis and
extracting intensity functions and experimental MCF profiles (for the
purpose of detecting and eliminating inter-version dependence during
the acceptance testing, not assumed a standard procedure in this
experiment) you may not wish to keep trace.<name>all, vect.<name>all and
binrep.<name>all files. The ter1.<name>all file contains a compressed
overview of the executed code blocks (all begining with 0.---| have not been
executed and you should find out why). You will not generate the
trace, vect and ter1 files, nor keep binrep if you do not use the -x option.
You may also wish to dispense with <name> and <name>.p files which are
the executable harness+rsdimu and source harness+rsdimu respectively.
We would recommend that you save at least the test.<name>all and
the error.<name>all files.

Any communication (questions and answers) received prior to corrected
program version are also saved into the "active" newcode sub-sub directory
as "q1", "a1", "q2", "a2" etc.

IV.

Upon receiving a message with the location of the latest corrected version, and of the correction/change report(s), cd to accept/:

* Create a new subsubdirectory in the appropriate program subdirectory
  e.g. ncsuB3.i location and change report have just been received

```
        mkdir newcode/ncsuB/v3
```

* Save the location/change report message into v<number>, e.g.
  from inside the mail:

```
        s <#> newcode/ncsuB/v3/correction_report

        where <#> is the number of the mail message on your h-list.
```

* Then (<path> points to maintenance team location the code):

```
    cp <path>/ncsuB3.i newcode/ncsuB/v3/ncsuB3.i

    cp <path>/ncsuB3.i ../code/ncsuB3.i

    fts_accept ncsuB3.i ncb3 all -c -x > test.ncb3all&
```

Now repeat the previous steps depending on the results of the test run, i.e. run a fts_correq if necessary, move results of the run into, for example v3 etc. Use appropriate university name and version numbers.

```
******************************************************************************
Notes:
******************************************************************************
```

It is expected that the maintenance team makes a single error correction that was requested by the correq.<name> report and sends back to you a mail message giving the location in their directories of the new and corrected code version, the new version number and one (or more if several changes had to be made to correct an error) error correction report(s). Save the received location message and the correction report into the "active" newcode sub-sub directory as "correction_report", e.g.

```
                    newcode/ncsuB/v2/correction_report
```

You procede then to pick-up the new version of the code and copy it into appropriate newcode sub-sub file, and ../code file (you may wish to use pointers/links to save space instead). Check that they send you the code and the report with an appropriate version numbers everytime.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>                                                                          >
> Check the error report they send against the test report you have!!!     >
> If there is any indication at all that the difference may be due        >
> to an error in the "golden" code (i.e. supplied expected answers)        >
> freeze all testing and immediately inform ATS distribution site,         >
> i.e. NCSU (see fts86/ReadMe for address, phone etc.).                    >
>                                                                          >
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

It is extremely important for the success of the experiment that you keep not only the final, corrected version of each program, but that each intemediate version submitted for acceptance testing is saved and tagged with an appropriate version number and information about the changes/corrections (using the provided change form). It is expected that programmers will correct one error at a time (and should not be given requests for more than one correction at a time), so that we can keep track of the influence particular errors had on the overall system failure probability etc.

The "certify" directory contains code and files that would be sent to each maintenance/certification team. It contains a basic rsdimu driver (to avoid interface problems) and instructions on its use. It also contains a sample input and output, and an electronic error report file. Whenever a change is made in the code it is expected that the programmer will record using this report. The new version of the code and the error and change report copy are both returned to the experimenters.

It is essential that each program be given a version number and associated with it the date of it creation. Every time a program is corrected its version changes and should be recorded in the correction report, as comments in the code itself, and should reflect in the file name for the new code (as kept in the "code" directory, and in the "newcode" directory in accept/).

****

Fault-Tolerant Software Experiment

PROCEDURE FOR CERTIFICATION TESTING

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

It is assumed that the communication between the experimenters and the maintenance personnel will be via electronic mail. It is further assumed that the experimenter has full access to maintenance personnel files, but the reverse is not true. It is also assumed that the acceptance testing is performed in csh in UNIX 4.2/4.3BSD, or Ultrix 1.1/1.2 environments running on VAX hardware (else see nonVAX_host/ directory).

I.

The testing begins by placing the code which is to be tested into the code/ directory. Enter the accept/ directory and start the initial round of testing by executing the fts_certify shell script. For example:

```
fts_certify ncsuD7.i ncd7
```

or

```
fts_certify ncsuD7.i ncd7 > certify.ncd7&
```

The latter form should be used if you wish to run in the background.

NOTE: YOU CANNOT RUN TWO fts_certify JOBS FROM THE SAME DIRECTORY AT THE SAME TIME (EVEN IN BACKGROUND). THE SYSTEM WAS NOT DESIGNED FOR THAT AND YOU CAN END UP WITH A MESS. YOU CAN, HOWEVER, KEEP TWO DIRECTORIES, SAY ACCEPT1 AND ACCEPT2 AND RUN AN fts_certify FROM EACH OF THEM WITHOUT INTERFERENCE.

fts_certify calls fts_accept with all.dat testset and -c option. Output is automatically routed into file test.<name>all. This file will then contain information about the acceptance test run, and will be used as the basis for forming a correction request file correq.<name>.

The run should either result in error messages (exit code <= 8) or should complete successfully (exit code 11). When the job ends check the test.<name>all file carefully.

* No compiler errors or missing voter call problems (exit status > 5).

* Fatal execution time errors exit status = 7.

* Differences detected from expected output exit status =8.

II.

If the test run ends with any status but exit(11), i.e. complete success, an error correction request will be produced for the maintenance team. The correction request will be in correq.<name>, and the test cases that caused up to the first 20 failures will be in errdata.<name>.

Check the content of correq.<name> and mail it to the maintenance team working on the <name> code (in examples: ncsuD7.i and higher versions, i.e. <name>=ncd7, or ncd8 etc).

Mail or copy directly into the directory of the maintenace team file errdata.<name>. The format of the data in this file is suitable for direct use by their "driver" program, so that they can do their own testing.

```
************************************************************************
*                                                                      *
*                           W A R N I N G                              *
*                                                                      *
* Your system may have a byte limit on mail messages that can be passed *
* through it (e.g. 100,000 bytes). In that case you may find that      *
* your correction report may be too large, and may become truncated    *
* by the e-mail system. It is safer to send only short messages and    *
* to transfer long files directly into certification team's directory  *
* using cp.                                                            *
*                                                                      *
* e.g. 20 failures in ncsuD7.i generate a correq.ncd7 request file of  *
*      about 4000 lines of code (about 170,000 bytes).                 *
*                                                                      *
************************************************************************
```

You may also have situations where you need to send non-standard messages as part of the correction request. For example, people may send you code and reports with incorrect or inappropriate version numbers. In situations like that create and insert the message at the begining of the correq.<name> file, just after the standard initial paragraph.

III.

The following procedure describes program and data version management without RCS or SCCS. You should read it regardless of the management procedure you will use so that you can decide what to save/store.

Create a subdirectory that will hold the starting, and all subsequent versions for a particular program(mming team). For example:

```
    mkdir newcode/ncsuD
```

make a sub-subdirectory for the current program version:

```
    mkdir newcode/ncsuD/v7
```

and move all the files you want to keep into that sub-sub directory, e.g.

```
    mv *ncd7* newcode/ncsuD/v7
```

Unless you are interested in doing correlation analysis and extracting intensity functions and experimental MCF profiles (for the purpose of detecting and eliminating inter-version dependence during the acceptance testing, not assumed a standard procedure in this experiment) you may not wish to keep trace.<name>all, vect.<name>all and binrep.<name>all files. The ter1.<name>all file contains a compressed overview of the executed code blocks (all begining with 0.---| have not been executed and you should find out why). You will not generate the trace, vect and ter1 files, nor keep binrep if you do not use the -x option. When using fts_certify this is default in or der to reduce program testing time and sve storage.

You may also wish to dispense with <name> and <name>.p files which are the executable harness+rsdimu and source harness+rsdimu respectively. We would recommend that you save at least the test.<name>all and the error.<name>all files.

To save space you can save only the difference in the code between the starting version and the new version (e.g. ncsuD7.i and ncsuD8.i, or ncsuD7.i and ncsuD9.i). For that purpose use the diff processor (read DIFF(1) manual).

You can further reduce storage space by "compressing" all the saved files using compress, or any other code compression processor available on your machine.

Whatever the scheme, make sure that you can rebuild the starting files and versions.

Any communication (questions and answers) received prior to corrected program version are also saved into the "active" newcode sub-sub directory as "q1", "a1", "q2", "a2" etc.

IV.

Upon receiving a message with the location of the latest corrected version, and of the correction/change report(s), cd to accept/:

* Create a new subsubdirectory in the appropriate program subdirectory e.g. ncsuD8.i location and change report have just been received

```
mkdir newcode/ncsuD/v8
```

* Save the location/change report message into v<number>, e.g. from inside the mail:

```
s <#> newcode/ncsuD/v8/correction_report
```

where <#> is the number of the mail message on your h-list.

* Then (<path> points to maintenance team location the code):

```
cp <path>/ncsuD8.i newcode/ncsuD/v8/ncsuD8.i

cp <path>/ncsuD8.i ../code/ncsuD8.i
```

alternative for latter (saves space):

```
{
    cd ../code
    ln -s ../accept/newcode/ncsuD/v8/ncsuD8.i  ncsuD8.i
}

    fts_certify ncsuD8.i ncd8 > certify.ncd8&
```

Now repeat the previous steps depending on the results of the test run.
Use appropriate university name and version numbers.

```
******************************************************************************
Notes:
******************************************************************************
```

It is expected that the maintenance team makes a error corrections for up to the first 20 reported failures that were requested by the correq.<name>. Certification (maintenance) team is supposed to mail back a message giving the location in their directories of the new and corrected code version, the new version number and one (or more if several changes had to be made) error correction report(s). You may if you wish use hardcopy correction reports, but there is a danger that the reports may evenuatly get separated from the code and corrections to which they refer. Furthermore if kept in electronic form it may be easier to analyse them.

Save the received location message and the correction report(s) into the "active" newcode sub-sub directory as "correction_report", e.g.

```
                    newcode/ncsuD/v8/correction_report
```

You procede then to pick-up the new version of the code and copy it into appropriate newcode sub-sub file, and ../code file (you may wish to use pointers/links to save space instead). Check that teams send you the code and the report with an appropriate version numbers everytime.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>                                                                             >
> Check the error report tems send against the test report you have!!!        >
> If there is any indication at all that the difference may be due            >
> to an error in the "golden" code (i.e. supplied expected answers)           >
> freeze all testing and immediately inform ATS distribution site,            >
> i.e. NCSU (see fts87/ReadMe for address, phone etc.).                       >
>                                                                             >
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

It is extremely important for the success of the experiment that you keep not only the final, corrected version of each program, but that each intemediate version submitted for acceptance testing is saved and tagged with an appropriate version number and information about the changes/corrections (using the provided change form).

The "certify" directory contains code and files that would be sent to each maintenance/certification team. It contains a basic rsdimu driver (to avoid interface problems) and instructions on its use. It also contains a sample input and output, and an electronic error report file. Whenever a change is made in the code it is expected that the programmer will record it using this report. The new version of the code and the error

```
   and change report copy(ies) are returned to the experimenters.

   It is essential that each program be given a version number and
   associated with it the date of its creation. Every time a program is
   corrected its version changes and should be recorded in the correction
   report, as comments in the code itself, and should reflect in the file
   name for the new code (as kept in the "code" directory, and in the
   "newcode" directory in accept/).

*****************************************************************************
```

Fault-Tolerant Software Experiment

Instructions for Certification Teams

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/12-May-87

Welcome to the FTS certification project. This is a NASA sponsored experiment in Fault-Tolerant Software. Your part will be to find and correct any bugs in the software, under controlled conditions. The software means one of the twenty programs that were produced during the first phase of this experiment.

You should be in possession of the following documentation (if you are not please see your experiment supervisor).

RSDIMU specification (version 3.2/10-Feb-87)

RSDIMU specification (version 2.1/19-Sep-85)

You should also have a directory called "certify" sent to you by the FTS experimenters. In this directory you have the code for the driver, instructions for its use, and some test data. All file that are part of this testing package begin with 'fts_' and should not be modifed by you except under special circumstances, and after consultation with your experiment supervisor. You can of course copy them and then modify them at will. This is not advised.

You will be receiving messages about needed corrections via e-mail. You should read them, correct the program to the best of your abilities, and test it. The data on which the program failed will either be attached to the message or will be sent to you separately. You can then use this data to test your code.

The message with correction requests is expected to be of the form

correq.<name>

where <name> is the code for the program you are correcting (university code followed by the team code and program version number). The data for the cases you failed (suitable for use with your system are expected to in a file of the form

errdata.<name>all

Once you are satisfied that you have found the fault that is causing the error(s) you were requested to correct, you should fill out (make a copy) the error_report and send it to your supervisor's e-mail address. Your site may also require you to fill in and submit a hard copy of the report. In the same message you should also tell us what is the current version of your program and in which file one can find it (we shall need copies of your corrected programs so do not change them once you have sent a message that one is ready for pick-up).

The program which you have just
finished correcting must be in a file called <unam>XX.vYY,
where <unam> is the agreed upon abbreviation for the university at which the
code was originally produced (e.g. ncsu, ucla, uiuc, uva), XX
stands for the letter and number associated with your program code,
and YY is the current version of your code (you begin by incrementing the number
in XX by one). For example C6, i.e. ncsuC6.v07 means that you have updated
ncsuC6 to version 7). You should also learn to update the version number
in the program header in the style in which it is already there.
If it is not part of the code you should add a comment header with the
version number and date (e.g. ncsuC6.v07/15-Jul-86). A sample header is
shown in fts_driver.p code.

If in doubt please ask about details.

Please bear in mind that the original specifications have been changed
and that the latest version (the one you have) may require you not only
to correct existing code, but also to add to it (for example missing calls to
voter routines).

File ReadMe_data contains a description of the test cases that are being
used to test your code. You should use this list in conjunction with
the correq.<name> report to locate and identify errors.

Please read the documentation you have received for the experiment.
Note that you should keep all communications concerning the
program you have been given between yourself and the experimenter,
and should communicate through electronic mail

****

e-mail address is: fts

****

Please feel free to ask e-mail questions about any part of this experiment.

Note the following rules and guidelines:

1. Do not change protections on any work-related files.

2. Communications are restricted during this experiment as follows.
   You may not discuss any aspects of your work in this
   job with other programmers. Any work-related communication
   between you and the Professor is to be conducted via UNIX mail. We
   require this so in the event of an error or ambiguity in the
   specifications, or some other significant event, all students
   may be sent a copy of the mailed question and its answer.

3. Every day you work you must log onto your UNIX account. In this way you will
   receive in a timely manner all mail concerning answers to
   questions, any updates in the specifications etc.
   You should also fill in a time-sheet once a day.

4. Your Professor will read the UNIX mail once early in the morning and again in the late evening (Monday through Friday). All questions should be directed to him/her.

5. It is your responsibility to read about UNIX tools you are not familiar or comfortable with.

6. Once a week, on Friday, you should submit a weekly progress report, describing the work you did during the week (number and type of errors you have corrected, any problems you have encountered running the driver harness, hardware problems, the total time you have spent working on the project during the week, whether reading or using the computer, if reading you should specify what and which part of the specs or which error prompted you to that action, etc.). Report is to be submitted via e-mail.

Good luck

Fault-Tolerant Software Experiment

Data

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

All the testcases in this release of the system comply with the specification version 3.2/10-Feb-87. A test case entry consists of an input record, and an output record. The latter contains what is believed to be the correct answer to the input record according to the 3.2 specs, and as generated/given by gold3v2.i.

Your code will be tested with a set of extremal and special value (ESV) test cases (796 of them) followed by 400 random test cases.

The ESV test cases are based on an initial random case which was then modified step by step to check a particular function and/or option. In the following overview of the ESV data only the principal features of the data sub-classes are given, along with the principal variables that were changed at any one time. Changes are given with respect to a base test case (usually #1).

1. A general random test case, DMODE=0 (RTI, foofl.dat), votecontrol=0. Checks that all the voters are off.

2-7. Test cases checking voter placement. Voters in the fts_harness are activated via VOTECONTROL one at the time. VOTECONTROL activation order 1, 3, 2, 4, 8 16.

Votecontrol and their activation results are detailed below:

1 : Sets the Linnoise values of first 4 sensors to true.
3 : Sets Linoffset value of first sensor to 0.0
2 : Sets linout value of first sensor to BADDAT;
4 : Sets SYSSTATUS to false.
8 : sets the estimate values of acceleration values to large values of 99999.0
16 : Sets garbage values in display output values.

8 General test case, (base 1) DMODE = 88.

9-19 base = case #1, systematic changes in DMODE testing for principal display modes (0,21-24,31-33,1,2,99).
Check for various display modes which do valid displays, and the boundary display modes.

20-27 Check for mod 4096 (all chans), base = #1, offraw:selected values are increased with 4096 or 8192 to check if only lower order bits are being used.

28-52. base = #1, changes in DMODE and LINFAILIN, checking for different "blank" displays, specific failure display formats, and failures of one sensor (28-37), whole faces (two sensors, 38-43), and various combinations of four failing sensors (44-52), with one instance of eight failed sensors (50).
The sensors are failed on input, to check for I display.

53-85. base = #1, random activation of different display modes continues (to ultimately test all values 0-99 by the end of the ESV set).
Noise on calibration (OFFRAW) in steps of +/- 6, +/-12, +/- 18 and +/- 24. Case 57 test has LINSTD=8, DMODE=1 and noise on calibration channel 1 of +/- 24 (8x3=24).
Also checked are the display failure formats for LINNOISE values, and the correct use of variable LINSTD, and correct computation of calibration noise levels.
6 and 24 were chosen because these were the boundary cases for noisiness for the linstd values chosen.

87-110 changes in LINSTD (9, 2,1 with +/- 24 on i/p channels) to check correct use of LINSTD variable and sensitivity of the calibration procedure.

86, 111-149. changes in RAWLIN, DMODE, LINFAILIN, various combinations of failures on input, noise and edgevector failures, base = #1.
Values in RAWLIN are so changed as to reflect an assured failure in edgevector test, so that there are no ambiguities left.
The values of DMODE are again chosen to test the display failure format. The failures are combined with failures on input, to see if the edgevector tests are properly employed.

150-151 Large changes in misalign [i,6] field, only the sixth axis was chosen for contamination because according to the latest specifications that is the only angle not used in the rsdimu procedure. It use significantly changes output values only if its value is much larger than normal. Changes in the values of other angles will not provided new information.

152-392. Test cases checking for the minimal sensor noise levels for failure declaration. Cases 152-365 no prior failures. Cases 366-392 prior failures on one and two faces.
These test cases test the sensitivity requirements that all three edges fail the edgevector test before a failure is declared. False alarms are raised when only one or two edges fail.
The normal value for the triplet threshold is 49 counts away from the correct figure for no prior failures on the rsdimu. The threshold values will change with the number and place of previous failures.

393-796 CRA proposed test cases with various combinations of sensors failed on input and up to one additional sensor failed in the edge vector test.

```
         56 test cases with 1 sensor failed on input.
        168 cases with two sensors failed on input.
        120 cases with 3 sensors failed on input.
         30 cases with 4 sensors failed on input.
          8 cases with 7 sensors failed on input.
          and the rest are other combinations.

Test cases numbers higher than 796 refer to random test cases.
```

Fault-Tolerant Software Experiment

DRIVER FOR THE RSDIMU CERTIFICATION

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

**** You do not need to read this if you will be using fts_compile and
fts_execute macros.

fts_driver.p is a Pascal driver program to run your rsdimu procedure.You
may make modifications to suit your tastes, but it is adequate in its present
form. To compile it you have to include your file containing rsdimu procedure
in the place provided in the source code. (It has to have a .i suffix to
run successfully). Also you have to make a call to rsdimu procedure at
the place designated.

The compiler command would be:

```
pc -C -s -o driver fts_driver.p
```

-s option would circumvent any problems you may encounter due to mixed
letter cases and non-standard i/o handling.

The executable module is created in file "driver", which can be run
as a shell command.
The driver expects the testcase input in a format as shown in the file
"fts_errdata.sample". The output, after a successful run of the driver,
is in fts_sample.out. Note driver is interactive.
If you wish to generate your own input data you will need to use the
"No_output_data" option.

Note that there are several parameters which are special and are
not part of the rsdimu variable/parameter set and are not given in
the specs. These variables appear at the beginning of the fts_errdata.sample
file. The rsdimu parameters begin with 15.0000 for obase. If you wish to
use the fts_driver.p on its own and without golden data then you need
to retain only the line before 15.0000 (votecontrol, case number).
Votecontrol serves to control special voter routine actions (whether
a particular voter changes the values of it parameters or not). It is used
solely for testing placement and use of the voter routines. You need
to leave it as is for regression testing of your code after correction.
You may experiment with it if you wish to build you own test sets.
You do not have to worry about it in the rsdimu code, the variable
is taken care of in the driver code.

The other parameters control the comparisons with golden answer and
you do not need to use them, unless you provide full format of the
file (with dummy golden answers for example).

Fault-Tolerant Software Experiment

Testing RSDIMU code

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

Using the fts_compile and fts_execute macros is simple. Run them without any parameters to obtain the description of the paramters you need. The following sample should help.

It is assumed that you have received correq.ncd6 and errdata.ncd6all from your experiment supervisor.

To compile program ncsuD7 which is (let's assume so) in your certify file, and is the program you have just corrected run

```
fts_compile ncsuD7 7 > c.7&
```

When the run finishes check c.7 for compilation errors etc. If ok proceed (rsdimu.7 will contain driver+ncsuD7 executable code).

```
fts_execute 7 errdata.ncd6all ncsuD7 > x.7&
```

When job finishes check x.7. If there still are differences from the expected outputs go back and correct your code once more, otherwise submit error_reports and the new code to your supervisor.

Make use of the correq.ncd6 and ReadMe_data.

Fault-Tolerant Software Experiment

DATA FOR THE ACCEPTANCE TESTING

RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

All the testcases in this release of the system comply with the specification version 3.2/10-Feb-87. A test case entry consists of an input record, and an output record. The latter contains what is believed to be the correct answer to the input record according to the 3.2 specs, and as generated/given by gold3v2.i.

The test cases are supplied in Pascal readable files. The format can be found in the fts_prnt.p source code in accept/. The format in which the test cases are given is suitable for use with the accept/fts_accept. Use of the system in non-Vax and UNIX-like environments is described in nonVAX_host directory, data may be re-generated using code suplied in the "generators" directory. The latter action is should not be undertaken without consultation with the ATS distribution site (NCSU).

If you wish to print out all, or some, of the test cases use accept/fts_listdata.

If you wish to compare (difference) test cases use fts_diff.

This set of test cases was designed and generated for acceptance testing of the rsdimu code. It consists of a group of 796 extremal/special value (ESV) test cases and a group of 400 random test cases. There are four files of data:

all.dat	- ESV test cases, followed by random test cases (randomNCSU.dat, then randomCRA.dat).
esv.dat	- ESV test cases, only (796).
randNCSU.dat	- random test cases, only (uniform sampling, 200).
randCRA.dat	- random test cases, only (shaped sampling, 200).

A successful pass through all the test cases gives an estimated lower limit on the reliability of the rsdimu code of about 0.992 (valid for the employed sampling profile).

The all.dat set should provide 100% block coverage of the rsdimu code. If this is not the case (running fts_accept with -x option will give the coverage info), one should very carefully examine the tested code in places where coverage was not provided. The nature of the rsdimu problem, and the specifications, is such that a thorough programmer can provide for situations and functions which are not explicitly handled in the specifications (e.g. singular matrices, large changes in the slope constants leading to large raw acceleration values). Redundant code of the type that cannot be excited according to the current specifications, but could possibly be needed under exceptional circumstances, should be tested by the programmers providing it. They should also provide test cases for these situations (if possible). Alternatively they should provide a written explanation of the cirumstances and reasons for including that particular code. The golden program gold3v1.i, for example has 5 blocks handling display of extremal input acceleration values (>10g) which are not tested by the current acceptance data set since such large input values are outside the conversion range of the provided equations.

The coverage figures should be considered only in the last stage of the acceptance testing, i.e. when all.dat cases have been passed without a failure, and all the corrections requests have been implemented (e.g. after the final regression pass through all.dat).

The ESV data set is further described in the ReadMe_esv file, and the random data sets are described in the ReadMe_random file.

Fault-Tolerant Software Experiment

THE ESV DATA FOR THE ACCEPTANCE TESTING

RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

The data file esv.dat contains 796 extremal and special (ESV) test cases. The test cases were designed to provide full functional coverage of the RSDIMU specifications v3.2/10-Feb-87.

The test cases are based on an initial random case which was then modified step by step to check a particular function and/or option. In the following overview of the ESV data only the principal features of the data sub-classes are given, along with the principal variables that were changed at any one time. Changes are given with respect to a base test case (usually #1).
Listing of all or some of the test cases can be obtained by running accept/fts_listdata. Difference between test cases may be examined using fts_diff.

1.   A general random test case, DMODE=0 (RTI, foofl.dat), votecontrol=0. Checks that all the voters are off.

2-7. Test cases checking voter placement. Voters in the fts_harness are activated via VOTECONTROL one at the time. VOTECONTROL activation order 1, 3, 2, 4, 8 16.

Votecontrol and their activation results are detailed below:

1 : Sets the Linnoise values of first 4 sensors to true.
3 : Sets Linoffset value of first sensor to 0.0
2 : Sets linout value of first sensor to BADDAT;
4 : Sets SYSSTATUS to false.
8 : sets the estimate values of acceleration values to large values of 99999.0
16 : Sets garbage values in display output values.

8   General test case, (base 1) DMODE = 88.

9-19   base = case #1, systematic changes in DMODE testing for principal display modes (0,21-24,31-33,1,2,99).
Check for various display modes which do valid displays, and the boundary display modes.

20-27 Check for mod 4096 (all chans), base = #1, offraw:selected values are increased with 4096 or 8192 to check if only lower order bits are being used.

28-52. base = #1, changes in DMODE and LINFAILIN, checking for different "blank" displays, specific failure display formats, and failures

```
          of one sensor (28-37), whole faces (two sensors, 38-43),
          and various combinations of four failing sensors (44-52),
          with one instance of eight failed sensors (50).
          The sensors are failed on input, to check for I display.

   53-85. base = #1, random activation of different display modes continues
          (to ultimately test all values 0-99 by the end of the ESV set).
          Noise on calibration (OFFRAW) in steps of +/- 6, +/-12,
          +/- 18 and +/- 24. Case 57 test has LINSTD=8, DMODE=1 and noise
          on calibration channel 1 of +/- 24 (8x3=24).
          Also checked are the display failure formats for LINNOISE
          values, and the correct use of variable LINSTD, and correct
          computation of calibration noise levels.
          6 and 24 were chosen because these were the boundary cases
          for noisiness for the linstd values chosen.

   87-110 changes in LINSTD (9, 2,1 with +/- 24 on i/p channels)
          to check correct use of LINSTD variable and sensitivity of
          the calibration procedure.

 86, 111-149. changes in RAWLIN, DMODE, LINFAILIN, various combinations of
         failures on input, noise and edgevector failures, base = #1.
         Values in RAWLIN are so changed as to reflect an assured failure
         in edgevector test, so that there are no ambiguities left.
         The values of DMODE are again chosen to test the display
         failure format. The failures are combined with failures on input, to
         see if the edgevector tests are properly employed.

 150-151  Large changes in misalign [i,6] field, only the sixth axis
          was chosen for contamination because according to the latest
          specifications that is the only angle not used in the
          rsdimu procedure. It use significantly changes output values only
          if its value is much larger than normal. Changes
          in the values of other angles will not provied new information.

 152-392. Test cases checking for the minimal sensor noise levels for
          failure declaration. Cases 152-365 no prior failures. Cases
          366-392 prior failures on one and two faces.
          These test cases test the sensitivity requirements that
          all three edges fail the edgevector testi before a failure is
          declared. False alarms are raised when only one or two edges fail.
          The normal value for the triplet threshold is 49 counts away from
          the correct figure for no prior failures on the rsdimu. The
          threshold values will change with the number and place
          of previous failures.

 393-796  CRA proposed test cases with various combinations of
          sensors failed on input and up to one additional sensor
          failed in the edge vector test.
                   56 test cases with 1 sensor failed on input.
                  168 cases with two sensors failed on input.
                  120 cases with 3 sensors failed on input.
                   30 cases with 4 sensors failed on input.
                    8 cases with 7 sensors failed on input.
          and the rest are other combinations.
```

More detailed information about the ESV test cases can be obtained
by displaying the differences between a chosen base case (#1 usually)
and a series of other test cases. Utility shell script fts_diff,
based on the UNIX diff processor, is provided for this purpose.
By executing

```
               fts_diff esv.dat esvdiff 115 123 1
```

you can obtain, for example, in file esvdiff differences in the input
values of cases 115 to 123 with respect to test case #1 of the data file
esv.dat.

The CRA document regarding choice of random and ESV test cases was provided
as a separate item (not in electronic form) with release 2.0 of RSDIMU-ATS.

Fault-Tolerant Software Experiment

THE RANDOM DATA FOR THE ACCEPTANCE TESTING

RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

This set of 400 random test cases for rsdimu code is provided primarily for the purpose of estimating the lower limit on the reliability of the tested code, and as a check on the completeness of the ESV test cases. The test cases are completely independent, and no attempt was made to mimic a flight trajectory and the associated time correlation among the input variable values. Therefore any cumulative effects linked to time correlation or auto-correlation remain untested, here and in the extremal and special value (esv.dat) set.

The random data are provide in two sub-sets: randomNCSU.dat and randomCRA.dat.

****

randomNCSU.dat

Within the employed sampling domain the distribution of the generated input values is essentially flat. It contains 200 test cases.

In all cases the random data were generated using the random number generator provided with the Pascal comiler (pc, UNIX/Ultrix). Details of the mapping from the random numbers into the actual input variables are given below. More details will be supplied on request. The part of the code used to generate random input values is also enclosed (as fts_NCSUzzrand.i), and it derives in part from the RTI random test harness (September 85).

The input variables randomly sampled, or computed on the basis of randomly sampled values are:

offraw, linfailin, rawlin, misalign, normface, temp, phiv, thetav, psiv, phi, thetai, psii, dmode, linnoise, linfailout, scale0,1,2, obase, linstd and nsigt.

A more thorugh understanding of the random generation process and of the resulting input profiles can be gained by studying the fts_NCSUzzrand.i code.

The random set, randomNCSU.dat, consists of two hundred random test cases stratified into two sub-sets. The first one hundred test cases have the noise on sensors (rawlin) boosted by 200 counts everytime linfailout for a sensor is true. Thus the sensor noise level is guaranteed to exceed the sensitivity threshold of about 50 counts and the sensor should be recognized as failed. The second one hundred test cases, on the other hand, have the noise added as a uniform distribution between 1 and maxnoise-1 counts, and at half the uniform frequency for 0 and maxnoise, the latter value having been read in by the driver program. In this particular case maxnoise was 110, therefore the added noise was symmetrically centered around the threshold value of 55 counts.

It is important to note that random test cases are intended to run after all ESV test cases have been successfully negotiated. There are special situations and combinations of variable values that are covered in ESV test cases and not covered by the sampling domain used to generate present random test cases. Our experience with the random testing of rsdimu code is that the sensitivity of the random test cases to errors is very low. Unless very detailed partitioning is employed (better to use ESV cases in that case) detection capabilities of the random test cases to distinct errors saturate extremely quickly. After 2 to 10 random test cases the same errors are usually detected over and over again (if not removed). Once past 100 random test cases detection of new, different, errors becomes an almost negligible event, unless the random sampling profile is changed and tuned to the character of the already detected faults, or partitions not previously covered are sampled.

For all practical purposes the two sets of 100 test cases, are a single random set of 200 test cases, which if executed successfully, provides us with a lower limit for the rsdimu reliability (at the 95% confidence level) of about 0.985.

initial random seed for 1st 100 cases is: 777

initial random seed for 2nd 100 cases is: 1234567890

****

randomCRA.dat

The second sub-set, randCRA.dat, was generated on the basis of the CRA document TM8602/26-Aug-86.

> CRA random test cases are generated with the specifications provided in the CRA documents (especially for PHIV, PSIV and THETAV, NSIGT = 2..7 etc). The calibration noise is normally distributed, and the number of noisy sensors during calibration is exponentially distributed with a parameter of 0.18. The edge vector test can fail one additional sensor, with random noise of upto 200 counts. No sensors fail on input. Generation details can be found in fts_CRAzzrand.i

initial random seed is: 987654321

****

For all practical purposes the two sets of 200 test cases, are a single random set of 400 test cases, which if executed successfully, provides us with a lower limit for the rsdimu reliability (at the 95% confidence level) of about 0.992.

During the generation of the random test cases care is taken to examine
the obtained data and to eliminate cases where more than one sensor
fails in flight.

Fault-Tolerant Software Experiment

THE GOLDEN DISPLAY AND RSDIMU CODES

RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

There are 2 files in this directory viz. display.i, and gold3v1.i
The sources correspond to specification version v3.2/10-Feb-87,

display.i contains the display module extracted from the gold program.
It has been extensively tested as a stand alone module, and
gives results in accordance with the specifications v3.2.
It does not need any special declarations in the main program except for
those which are in the RSDIMU procedure assumed to be
globally available (only the type declarations.)
To use this procedure just use standard #include compiler option, and
the calling format

```
   display (DISMODE, DISUPPER, DISLOWER);
```

Use of the golden rsdimu follows the same rules as use of any other rsdimu
code, and is fully explained in the specs (see also certify/driver.p).