LH GRANT

IN-61-CR

319008

P. 41

Computer Science

Technical Report

Software Reliability Through Fault-Avoidance and Fault-Tolerance

Report #3 (3/1/90-8/31/90) on NAG-1-983

by

Mladen A. Vouk and David F. McAllister

North Carolina State University

Box 8206
Raleigh, NC 27695

Semi-Annual Technical Report Submitted to the

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

for research entitled

SOFTWARE RELIABILITY THROUGH FAULT-AVOIDANCE AND FAULT-TOLERANCE

(Report #3 on grant NAG-1-983)

from

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

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

Report Period
Beginning Date: March 1, 1990.
Ending Date: August 31, 1990.

Table of Contents

Project Progress Summary

In this project we have proposed to investigate a number of experimental and theoretical issues associated with the practical use of multi-version software as a tool for avoiding faults during system production and for providing run-time tolerance to software faults. In the period reported here we have worked on the following:

- We have continued studying back-to-back testing as an efficient mechanism for removal of un-correlated faults, and some common-cause faults. In particular we have investigated use of back-to-back testing for regression test and porting. (report attached)
- We are completing an empirical evaluation of a number of fault-tolerance models and strategies with respect to their performance in the presence of inter-version failure dependence. In particular, we are using RSDIMU software units,to evaluate N-Version Programming with majority voting and concensus voting, as well as Recovery Block, Concensus Recovery Block, and Acceptance Voting strategies. We find that consensus voting, Consensus Recovery Block and, in special cirumstances, Acceptance Voting offer reliability performance which is better and more stable than the classical majority voting approach. (report in preparation)
- A study of the cost-effectiveness of fault-tolerant software has shown that in the case when failures are independent, Consensus Recovery Block and Recovery Block are the only cost justifiable fault-tolerant techniques to be considered. Unless the voter is perfect, N-Version Programming does not compete cost-wise with the other two methods. However, the hybrid method, Consensus Recovery Block, which contains both voting and recovery block can provide considerable reduction in cost for a given system reliability over the other techniques. (report attached)
- We continued studying software reliability estimation methods based on non-random sampling, and the relationship between software reliability and code coverage provided through testing. Software testability modeling, based on control and data flow construct coverage, is in progress. A coverage based software reliability model will be developed and used as part of the reliability modeling process.
- We have completed and published a study of similar faults in RSDIMU software. (report attached)

This report describes the results obtained in the period March 1, 1990 to August 31, 1990.

1. Using Back-to-Back Testing for a Regression Test and Porting*

Mladen A.Vouk
Department of Computer Science, Box 8206,
North Carolina State University, Raleigh, NC 27695-8206
Tel: 919-737-7886, Fax: 919-737-7382, e-mail: vouk@cscadm.ncsu.edu

Abstract

The paper examines the use of back-to-back, or comparison, testing for regression test or porting. The efficiency and the cost of the strategy is compared with manual and table-driven single version testing. Some of the key parameters that influence the efficiency and the cost of the approach are the failure identification effort during single version program testing, the extent of implemented changes, the nature of the regression test data (e.g. random), and the degree and the nature of the inter-version failure correlation and fault-masking. The advantages and disadvantages of the technique are discussed, together with some suggestions concerning its practical use.

1. Introduction

Software testing and validation based on program execution may consume a large part of the development effort. Investigations of the reliability growth process have shown that the major part of any testing effort may be the identification of failures [e.g. Mus87]. Therefore, a testing strategy where failure identification is an automatic and low cost process has distinct advantages. In this context the development and testing of two or more functionally equivalent versions of a program against each other has considerable appeal because it offers a potentially very simple way of checking for the correctness of a very large number of test cases. One of the names[1] for the technique is back-to-back testing [Bis86, Vou90a]. Back-to-back testing can be very effective in detecting dissimilar software faults, as well as some classes of similar faults [e.g. Pan81, Bis86, BrK86, Vou88a, Kel90]. It can also be cost-effective [e.g. Pan81, Sag86, Vou88b].

An interesting variant of back-to-back testing is its application to testing of a program after modifications. This includes maintenance regression testing, and testing of software after conversion within[2] or between[3] languages, or after the code has been ported to one or more different host machines[4]. Use of back-to-back testing as part of an intra- or inter-language software conversion process is discussed in [Wol83].

* This work was supported in part by NASA grant NAG-1-983. Paper published in Proc. 7th International Conference on Testing Computer Software, pp xx-yy, 1990.

1 Some other names for the technique are comparison testing, diverse system testing, parallel testing, multiversion testing.

2 Intra-language conversion. For example, conversion from one FORTRAN dialect (compiler) to another.

3 Inter-language conversion. For example, from FORTRAN to C.

4 Of interest in applications that run on many platforms. Examples are numerical and other software libraries or systems (e.g. NAG, IMSL, SAS).

More recent examples of the industrial use of back-to-back testing are the AIRBUS software [Tra88], Intel's use of N-version programming as one of the many tools they used in testing the 80960 microprocessor [Val88], and use of parallel calculations by the SAS Institute when testing and porting software to different host machines [McG90]. An unorthodox run-time use of comparison testing comes from the "wily hacker" story were a discrepancy reported by three accounting programs running in parallel was the first indication of a break-in into a Lawrence Berkeley Laboratory computer [Sto88].

In section 2 of this paper we discuss some properties of back-to-back testing strategy. In section 3 we present two models for using back-to-back testing during regression test and porting, and then we discuss the economics of this approach. Summary is given in section 4.

2. Back-To-Back Testing

In its traditional form back-to-back testing strategy involves (independent) production of two or more functionally equivalent programs (or *versions*, or *components*). These programs are tested statically (e.g. code reading) and dynamically (designed/functional test cases, and random test cases). All programs are tested with the same input data and the outputs of all possible program pair combinations are compared. Whenever a difference is observed the problem is thoroughly investigated in all versions, and for all test cases where even one component answer differs. The fault search is usually initiated in all versions after the first warning has been received. If a fault is found, a correction is applied.

Back-to-back testing accomplishes two functions:

(i) it detects failures (and indirectly faults) dynamically, and

(ii) it detects faults statically.

The first function is accomplished through comparisons of execution results and indirectly leads to detection of faults. The second one is the result of the fact that back-to-back testing process involves more than just comparison of results. Once a potential failure warning has been received, the process requires searching for the faults in all versions, even if only one version disagrees with the rest. This encourages code reading and re-evaluation of algorithms by different debugging teams in all versions. The cycle repeats until, for example, back-to-back comparisons stop reporting differences for the employed test data sets, or until some target failure intensity has been reached. All answers have to be identical to within a tolerance if a "no failure" event is said to occur.

Theoretical [Eck85], and experimental [e.g. Sco84, Kni86] work indicates that independently developed version may not fail independently, although, again in theory [Lit87], it should be possible to achieve independence through use of sufficiently diverse development methodologies. At worst, all versions contain the same similar (common-cause) faults which make them fail coincidentally for all inputs. If the probability of coincident failures is significantly different from what would be expected by chance then we say that the observed coincident failures are *correlated* or *dependent*.

Back-to-back testing fails to detect a coincident failure of one or more versions if all programs return identical answers. This can occur, for example, if there is an identical fault in all versions because versions are copies of each other, or because all programmers have made exactly the same mistake, or because the output space is binary. In general, the response of functionally equivalent programs to similar faults may be identical or may be different. It is also possible that dissimilar faults from different components cause a coincident failure and give either different answers or identically wrong answers.

Because inherently back-to-back testing cannot distinguish between identical responses from all versions (whether they are right or wrong), when used during software development it should always be followed by other testing strategies aimed at the residual faults. However, when back-to-back testing is used for regression testing or porting then the problem of identical and wrong responses diminishes in importance because the testing has different goals.

2.1 Regression and Porting Test

The intention of regression and porting test is to check back on any changes, and make sure that the changes have not injected, and/or stopped masking, faults, or have not corrupted already tested functions and parts of the code. Change errors, if any, are likely to be random and result in failures that are mutually independent. Because generation of a new version of the code is implicit in any software modification or conversion, functionally "(almost-)equivalent" version pairs (or 2-tuples) are available at no extra cost. This means that the "new" and "old" versions of the code can often be run against each other to verify invariance of the the functions and responses that were not supposed to have been affected by the applied changes.

Sometimes it is possible to conduct regression testing using all of the data available for testing, but often, due to execution time, project schedule, computer storage, or similar constraints, it is necessary to limit the regression testing to a smaller subset of the test data. Whenever regression testing is limited to a smaller subset of the total data set there is always some doubt that the "important" test cases, which could reveal an inadvertently injected bug, are not part of the regression set. Selection of the subsets is not a trivial problem and a number of researchers have addressed this [e.g. Yau87, Har90].

There are indications that in some circumstances random data may detect more faults than more conventional structured, partitioned and special value testing [e.g. EhE88, Ham88, Kel90]. It is, therefore, it is desirable to supplement testing based on a designed (fixed or growing) test set with random test data. The problem is that, unless failures are self-reporting, it may be very expensive to regression test with random data because of time, storage, answer correctness, and similar problems. If the only failures of concern are self-reporting failures (e.g. system crash, or an obvious disruption of the computer service) a relatively simple acceptance test, or consistency check, may be sufficient to verify the correctness of the answers. On the other hand, if the correctness of the responses is less obvious, then a more elaborate, and often time consuming, scheme must be used.

Similarly, there is experimental evidence [e.g. ShL88] that monitoring of internal program states can considerably enhance fault detection efficiency. Again, time, storage and correctness problems can present a considerable deterrent to practical use of this approach in regression testing. Another problem may be the diminished flexibility of "fixed" data regression sets to changes in the operational input profiles. Comparison of the answers with an existing, progressively generated and growing, database of "correct" answers is a natural solution. But, the time needed for output verification (identification of failures) may still remain problem.

One approach that can help in solving at least some of the problems mentioned above is back-to-back testing. When an application has associated with it a set of test cases to which exact correct responses are available in a tabular (file) format, detection of a discrepancy with respect to that set of test cases is usually trivial and does not require back-to-back testing. But, a big advantage of testing successive versions back-to-back is that the input data, and the corresponding answers, do not have to be pre-stored but can be generated during the testing. Furthermore, the range and the profile of these test cases can be readily changed to accommodate a different operational profile without a (possibly) costly re-generation of the regression data base. For example, Kelly and Murphy [Kel90] note that in their back-to-back testing experiment "The complicated operational stress tests would have been impossible to perform if the output had to be manually verified against the specification".

Another obvious advantage of using back-to-back regression testing is that a very large number of variables and intermediate states can be monitored relatively cheaply. This should increase sensitivity of the testing to any anomalies introduced or revealed during the modifications. Furthermore, probing of intermediate states and classification of the expected outputs according to whether a difference would, or would not, be observed with respect to the earlier version can yield useful information about the expected and actual coupling of, and dependencies within, the code (c.f. perturbation or mutation testing).

2.2 Failure Detection Efficiency

The probability, $P_D\{N,1\}$ (per test case), that back-to-back testing of N versions detects a failure can be expressed as [Vou88a]:

$$P_D\{N,1\} = 1 - [\gamma(N)\, P\{1+\} + P\{0\}] \tag{1}$$

where $P\{1+\}$ is the probability that "One or more of the N-versions fail coincidentally", $\gamma(N)$ is the conditional probability for the event "Answers from N versions are accepted as identical given that one or more of the N versions have actually failed", and $P\{0\}$ represents the probability that all k versions succeed (i.e. zero fail). The dependence among failures, if any, encompasses both the correlation due to the cardinality of the output space, and the correlation due to similar or common-cause faults. In the case of binary output space (cardinality 2; the answer is either correct or incorrect without distinction among possible incorrect answers, for example, a correct output is always 0, while an incorrect output is always 1) every failure will yield the same wrong answer regardless of whether the underlying faults are similar or dissimilar. In the case of similar faults the probability of identical and wrong answers will depend on the nature of the faults and the input data.

Let, for simplicity, p represent the per test case failure probability of a single version, and let this probability be the same for all versions. If there is no correlation at all $P_D\{N,1\} = 1-(1-p)^N$. If the correlation due to similar faults is zero but the output space is binary, then (1) becomes $P_D\{N,1\} = 1-[p^N+(1-p)^N]$. If there is only a single similar fault in all versions then $P\{1+\}=p$, $P(0)=(1-p)$, and $\gamma(N)$ must be less than one for the fault to be detectable by back-to-back testing. Ideally, back-to-back testing has the power to detect all failures the first time they occur ($\gamma(N) = 0$), or at least after the same failures have repeated several times ($\gamma(N) << 1$, $P\{1+\}>0$). It can be shown that the number of representative random test cases needed to guarantee detection of a failure at α confidence level is

$$T = \frac{\ln(1-\alpha)}{\ln(1-P_D\{N,1\})} \tag{2}$$

and that $P_D\{N,T\} = 1-[1-P_D\{N,T\}]^T$. Given a particular similar fault, the larger the failure correlation, the more test cases have to be executed before that fault is detected. In the extreme, when correlation is 100% ($\gamma(N)=1$, $P_D\{N,T\}=0$), the fault cannot be detected by back-to-back testing. It can also be shown that if there are no correlated faults which have failure span over all versions with $\gamma(N)=1$, then using more than about 4 versions results in rapidly diminishing returns in terms of the failure detection efficiency [Vou88a].

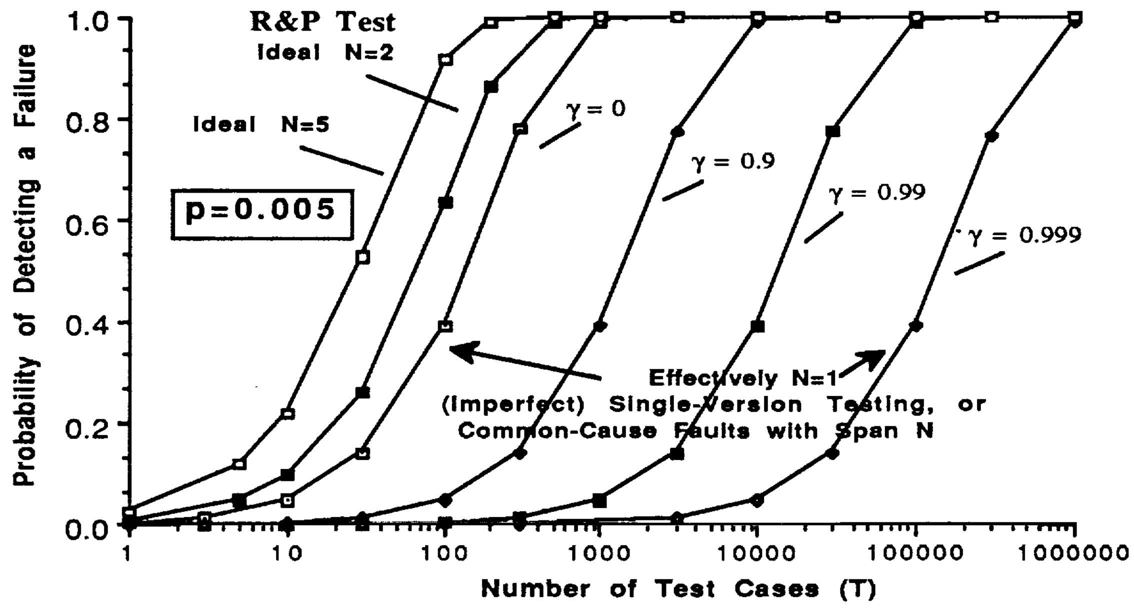


Figure 1. Failure detection efficiency of a typical Regression and Port (R&P) Test , and illustration of the influence of imperfect single-version regression test or inter-version failure dependence.

When back-to-back testing is employed to test for invariance after software changes (regression testing, conversion testing, host-change testing), it is very likely that the errors will be random and that the model assuming independent failures applies.

Figure 1 illustrates the gain that use of back-to-back testing offers with respect to single version regression testing based on random test data. For p=0.005, we plot the probability of failure detection using N versions against the number of test cases run. The curve marked "N=1" represents the ideal single version (e.g. manual) testing. The curve marked "Ideal N=2" was computed using equation (1) with independent failure model assumptions. It represents a typical regression testing situation ("old" vs. "new" version) where the effective failure detection properties are the same as that of two completely failure-independent versions [Vou88a, Vou90a]. The curve "Ideal 5" illustrates the gain of adding more versions which do not contain similar faults (more appropriate in host migration testing).

Note that the horizontal axis is logarithmic, and that the important difference between, for example N=1 and N=2 curves, is the vertical separation of the curves for a fixed number of test cases, or alternatively the horizontal difference in the number of test cases needed to achieve the same failure detection probability. The rest of the curves were obtained using equation (1) under the assumption of dependent failures. They show how increasing conditional probability of identical failures influences failure detection probability. In Figure 1 this conditional probability is denoted by $\gamma = \gamma(N)$. The γ-curves represent a general N-tuple in which all versions fail with identical responses with probability[5] $\gamma(N)$. We see that this conditional probability can be very close to one before the number of test cases needed to recover the detection efficiency starts rising so sharply that

[5] Alternatively they can approximate the effect of imperfect manual adjudication of the correctness during single version regression testing. In that case $\gamma(N)$ represent the probability that on manual inspection of the results the testers fail to detect an anomaly or discrepancy in the outputs despite the fact that the program has failed.

with automatic comparisons (back-to-back testing) it may not be practical to generate and execute them.

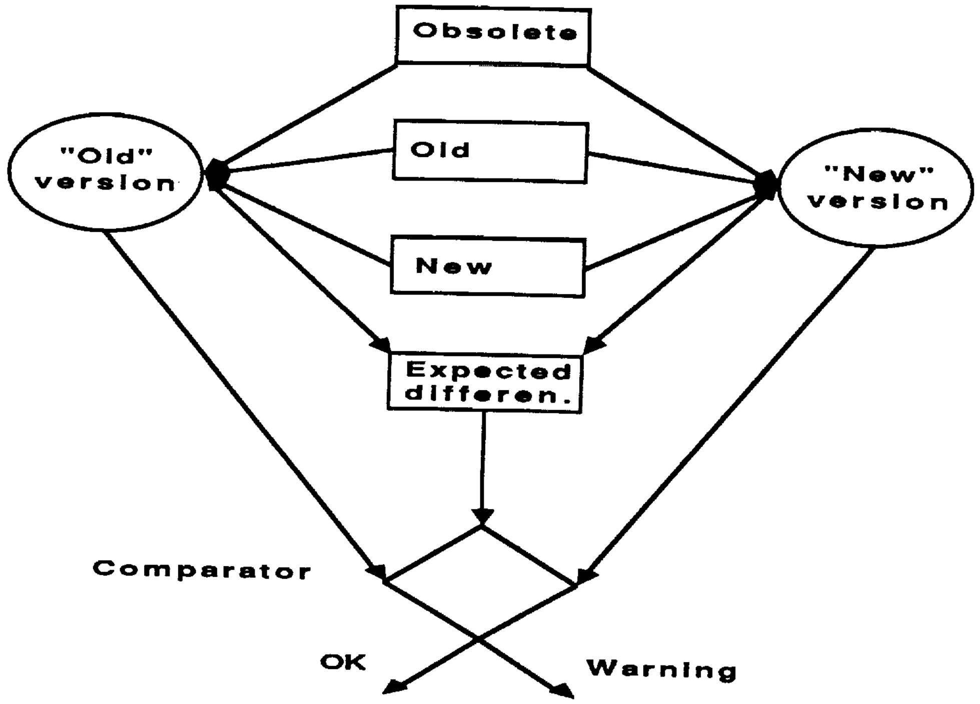


Figure 2. Single-stage back-to-back regression test.

3. Models and Cost

Regression testing can take place once, after all intended program changes have been implemented, or it can take place after every program modification step. We distinguish two models for back-to-back regression test and porting: single-stage and multi-stage.

3.1 Single-Stage Model

The single-stage model for back-to-back regression testing of "almost-equivalent" versions is illustrated in Figure 2. The circles depict two consecutive versions of the software, the squares the sources of data (e.g. files or data generators), and the diamond the answer comparator. The response comparisons can be made at almost any desired level; output only, module/function level, intermediate states, even line level. The nice part is that there are practically no problems with the insertion of the sampling probes because the code is not only functionally almost identical, but also structurally very similar. The differences, of course, exist in the modified parts of the code, and if they are too large, or have implications which are too expansive, a multi-stage approach may be considered (see section 3.2).

We assume that three "types" of regression data are available. An invariant ("old") set, which contains all the test cases which are still valid and completely unchanged following the program modification. A set containing "obsolete" test cases, cases which are no longer valid because of changed requirements, variable ranges, functionality of the code, environment, and similar. And, finally, a set of "new" or changed test cases which contains all the test cases that had to be modified, or were generated completely anew, to accommodate the changes in the functionality and structure of the code or to increase the test set. One file, "expected differences", contains a "list" of test cases (and responses) for which the differences between the "old" and "new" code versions would be expected to arise. This data needs to be generated, based on performed modification(s), prior to any regression testing. For example, if upward compatibility of versions is required because the changes are enhancements which should not affect previous performance (e.g. and extension of a communication protocol), then all of the "old" data set responses for key parameters should match (except for new variables), while the "expected differences" will derive primarily from the "new" data set.

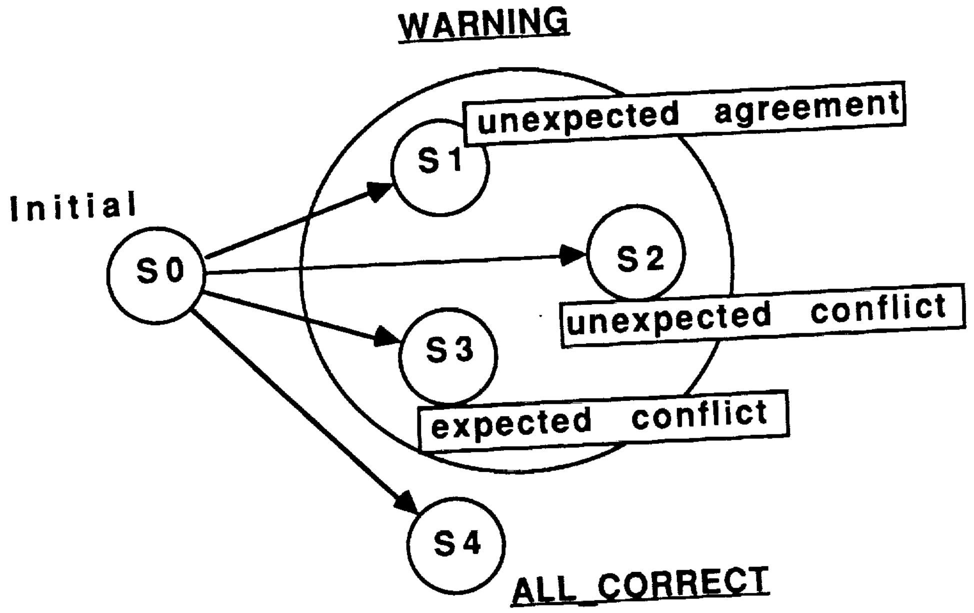
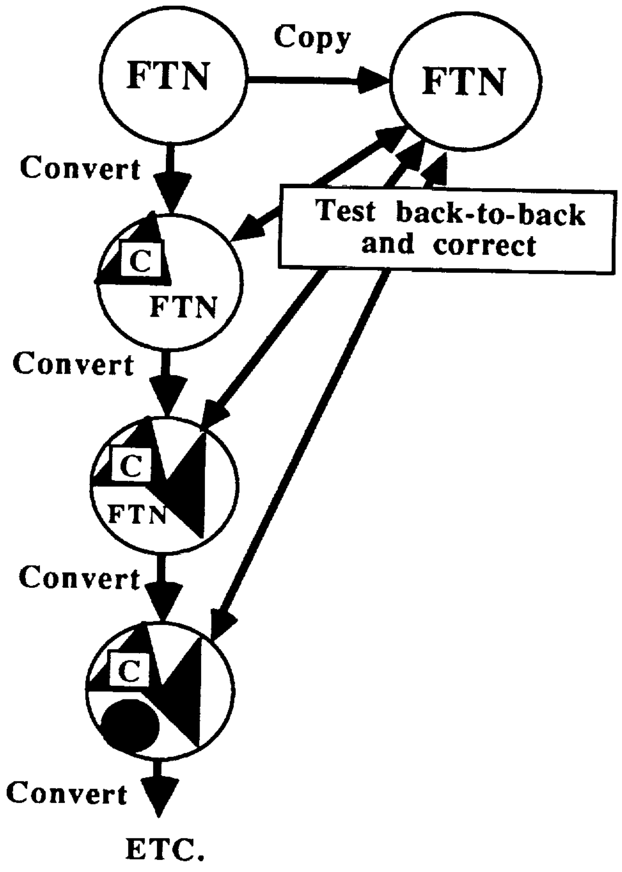


Figure 3. Transition states for (back-to-back) regression and change testing.

There are two general output states of the system. The system either issues a warning, or it accepts the comparison. We shall call the former a CONFLICT event, and the latter an AGREEMENT event. In principle, only unexpected differences or unexpected agreement between the outputs should raise an alarm. However, it is prudent to re-examine all outputs where differences arise unless the size and sign of the expected difference(s) is included in the data base. Unexpected disagreement(s) between the versions may be indicative of incompletely corrected faults, newly introduced faults, or old faults that are no longer masked owing to the implemented code changes. The question of tolerances, and false alarms should also be considered [Vou88a]. It is also possible that an expected difference in the response does not materialize. This should also be cause for alarm. The reason could be, for example, that the implemented change was not successful (although not detrimental), or that there is a fault in the test case, etc. The specific states that can occur are illustrated in Figure 3. Comparison of the results from the versions will result in a WARNING state whenever there is either a difference (conflict) of any kind, or an unexpected agreement. The ALL_CORRECT state implies that the new version behaves exactly as the old version did, i.e. only expected agreements have been observed.

3.2 Incremental (Multi-Stage) Model

If the differences in the modified parts of the code are extensive it may be more efficient to implement the changes incrementally and use back-to-back testing between the increments to verify their correctness. Incremental change implementation can help focus attention of testers, and may avoid masking of change induced faults. At each stage of the process single-stage back-to-back testing is performed using a partial or full set of test cases. Software processes such as intra- and inter-language conversion are well suited for this approach provided a mixed language programming environment is readily available.



Inter-Language Conversion

Figure 4. An example of a strategy that uses staged back-to-back testing.

Figure 4 illustrates an instantiation of a multi-stage back-to-back testing. It depicts incremental conversion of a FORTRAN program into C language. FORTRAN routines are converted to C only few at a time. After each conversion the "new" C version of the code is separately compiled and re-linked with the remaining FORTRAN code. Outputs from this mixed-language program are then compared with the original FORTRAN outputs to assert invariance of the program functionality.

3.3 Cost

In the case of new software development there are many conditions that need to satisfied before full
back-to-back testing becomes cost effective. In the case of a regression testing and porting, the
conditions are fewer and more easily satisfied. For example, software maintenance and conversions
produce an extra version as part of the software process in any case, so the overhead is mostly in the
effort needed to produce a driver for the versions, and a comparator of the results. If the versions are
in different languages the driver must also handle inter-language communication (directly or
indirectly). Most of the test cases and testing-related tools can be reused. This certainly reduces costs
associated with test case generation and with the development of a back-to-back testing harness.
However, if the failures are self-reporting, or fast table (file) look-up based adjudication of answer
correctness is available, then back-to-back testing may still be too costly.

The problem is discussed in more detail in [Vou88b, Vou90b]. The results indicate that the important
parameters are: the desired reliability (i.e. the number and quality of test cases that needs to be run),
and the ratio between the number of back-to-back tests cases that can be generated and evaluated for
the cost of every single version test case. This ratio must be between 2 and 10 before back-to-back
testing becomes cost effective, and there is always a certain minimal number of test cases below
which it is too expensive to construct a back-to-back testing harness.

4. Summary

The primary attraction of back-to-back testing lies in the fact that, once that testing environment has
been set up, a very large number of test cases can be generated (usually randomly), run, and
evaluated with minimal overhead in terms of human supervision. Testing of software after
modification is well suited for application of back-to-back testing strategy because production of
additional "almost-equivalent" software versions is implicit in the process of software change. This
eliminates a large fraction of the overhead costs normally associated with comparison testing, and
back-to-back regression testing is often cost effective. The technique is particularly effective when
random regression testing is used, a large number of intermediate states in monitored, there are
frequent changes in the operational profile and variable ranges between versions, or there are storage
problems but input data can be dynamically recreated. Back-to-back testing is not indicated if
program failures are self-reporting, a quick table-based correctness adjudication is available, a good
acceptance test or consistency check is available, or only a few test cases need to be re-run.

If used appropriately and in conjunction with other techniques, back-to-back testing can be an
excellent tool for aiding the development of high reliability software.

References

[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.

[BrK86] S. Brilliant and J.C. Knight, "Testing Software Using Multiple Versions", University of Virginia, Department of Computer Science, Report No. RM-86-07, 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.

[EhE88] Willa K. Ehrlich, and Thomas J. Emerson, "The Effect of Test Strategy on Software Reliability Measurement," 11th Minnowbrook Workshop on Software Reliability, July 1988.

[Ham88] D. Hamlet, and R. Taylor, "Partition Testing Does not Inspire Confidence," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 206-215, July 1988.

[Har90] J. Haartman, and D.J. Robson, "Techniques for Selective Revalidation," IEEE Software, pp 31-36, January 1990.

[Kel90] J.P.J. Kelly, and S,C, Murphy, "Achieving Dependability Throughout Development Process: A Distributed Software Experiment," IEEE Trans. Soft. Eng., Vol. 16 (2), pp 153-165, 1990.

[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.

[Lit87] B. Littlewood, and D.R. Miller, "A Conceptual Model of Multi-Version Software," FTCS 17, Digest of Papers, IEEE Comp. Soc. Press, pp 150-155, July 1987.

[McG90] S. McGrath, "Vendor Quality Assurance," North Carolina Quality Assurance Discussion Group, RTP, Meeting held on 4th April 1990.

[Mus87] J. Musa, A. Iannino, and K. Okumoto, "Software Reliability: Measurement, Prediction, Application," McGraw-Hill Book Co., 1987.

[Pan81] D.J. Panzl, " A Method for Evaluating Software Development Techniques", The Journal of Systems Software, Vol. 2, 133-137, 1981.

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

[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 .

[ShL88] T.J. Shimeall and N.G. Leveson, "An Empirical Comparison of Software Fault-Tolerance and Fault Elimination," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 180-187, July 1988.

[Sto88] C. Stoll, "Stalking the Wily Hacker," Communications of the ACM, Vol. 31 (5), May 1988., also "Stalking the Wily Hacker," Banquet Talk at COMPASS'88, July 1988.

[Tra88] P. Traverse, "AIRBUS and ATR System Architecture and Specification," in Software *Diversity in Computerized Control Systems*, U. Voges (ed.) , Springer-Verlag, Wien, Austria, pp 95-104, 1988.

[Val88] J. Valerio, "N-Version Programming," Network News COMP.RISKS FORUM, Vol. 7 (18), 8-July-1988.

[Vou88a] M.A. Vouk, "On Back-To-Back Testing," Proc. COMPASS '88, pp 84-91, June 1988.

[Vou88b] M.A. Vouk, "On The Cost of Back-To-Back Testing," Proc. 6th Annual Pacific Northwest Software Quality Conference, Lawrence and Craig, Inc., Portland, OR, pp 264-282, September 1988.

[Vou90a] M.A. Vouk, "Back-to-Back Testing, " Information and Software Technology, Vol. 32 (1), pp 34-45, 1990.

[Vou90b] M.A. Vouk, "Modeling Back-to-Back Testing," North Carolina State University, Department of Computer Science, Technical Report, 37 pages, 1990.

[Wol83] J.R. Wolberg, *Conversion of Computer Software*, Prentice-Hall, Englewood Cliffs, N.J. 07632, USA, 1983.

[Yau87] S.S. Yau, and Z. Kishimoto, "A Method for Revalidating Modified Programs in the Maintenance Phase," Proc. IEEE COMPSAC '87, CS Press, pp 272-277, 1987.

2. Cost Modeling of Fault-Tolerant Software*

by

David F. McAllister
Department of Computer Science
North Carolina State U.
Raleigh, N.C. 27695-8206

and

Keith Scott
IBM
PO Box 12195
Research Triangle Park, N.C. 27709

I. Introduction

In [1] Scott et. al. introduce data domain reliability models of several fault-tolerant software schemes including N-version programming , recovery block, and consensus recovery block. We extend these results by coupling them with a cost function and examine the results when reliability is constrained by cost and vice versa, i.e., the cost is constrained by system reliability. For tractability we restrict our development to 3 version systems and we assume that software failures are statistically independent. The reliability of a software module is the probability that it produces the correct result for a given input. Our notation will be consistent with our previous work. Let r_1, r_2 and r_3 be reliabilities of each version of a three-version fault-tolerant system, let B the the reliability of the acceptance test, let V be the reliability of the voter in N-Version Programming and let S be the system reliability. Then for N-version programming we have

$$S_{nvp}(r_1, r_2, r_3, V) = V(r_1r_2 + r_1r_3 + r_2r_3 - 2r_1r_2r_3) \qquad (1)$$

Recovery block becomes

$$S_{rb}(r_1, r_2, r_3, B) = B(r_1 + r_1r_2 + r_1r_2r_3 + r_2B - 2r_1r_2B + r_1r_3B + r_2r_3B - 4r_1r_2r_3B + r_3B^2 -$$

* This work was supported in part by NASA grants NAG-1-667 and NAG-1-983. Paper accepted for publication in ???.

$$2r_1r_3B^2 - 2r_2r_3B^2 + 4r_1r_2r_3B^2) \qquad (2)$$

while consensus recovery block is defined by

$$S_{crb}(r_1, r_2, r_3, B, V) = S_{rb}(r_1, r_2, r_3, B) + S_{nvp}(r_1, r_2, r_3, V) - S_{rb}(r_1, r_2, r_3,B)\, S_{nvp}(r_1, r_2, r_3, V) \qquad (3)$$

While the equations tend to become visually complicated, they are simple to treat using a symbol manipulation program such as Mathematica [4]. In addition, we will make some simplifying assumptions for tractability and understanding.

We will treat the optimization problem of minimizing system cost subject to the constraint that system reliability is fixed. In the following section we will discuss the choice of a cost function. In section III we will treat a special easily solved subcase of the constrained optimization problem . In section IV we treat a more general version of the optimization problem and solve it using Lagrange multipliers. In section V we summarize our results.

II. The Cost Function

We have assumed that cost increases exponentially as the reliability of a version approaches one. This follows directly from data domain reliability modeling [3] since adding a correct digit to the reliability estimate of a software module requires an order of magnitude more test cases if we use random testing. In addition, the cost function should have the line r=1 as a vertical asymptote. There are many choices for a cost function with the above properties and the techniques we propose here can be applied to others also. We have chosen the cost function for a single version to be

$$C(r) = \beta(1-r)^{-\alpha} + c$$

where r is the reliability of a version, and α,β, and c are positive constants which control the shape and location of the cost function. The constants β and c determine the intial or startup cost when r=0. Since our optimization results are independent of the constant c which appears linearly in the equation we will eliminate it from the definition of C henceforth. The final cost can be augmented by c without changing the optimal reliabilities.

The constant α controls the rate at which the cost increases as r approaches 1 and the constant β can be used to control the initial cost and differences between development and testing costs of each module. In the most general case, each version, the acceptance test and the voter can have different values of α, β with different reliabilities. To reduce the dimensionality of the problem we will resrict our attention here to the case when all versions have the same reliability r and α is the same for all components including the voter and the acceptance test. We will also assume that $\beta_i = \beta_r = 1$ for each of the modules. We will expect the β values of the acceptance test (β_B) and the voter (β_V) to be less than or equal to the β value of the versions because, in general, an acceptance test and a voter should be less complex and more easily tested than any of the versions.

We will examine the behavior of the cost function for different values of α and try to summarize our results and impart some intuition. We will treat two subcases:i) when B=V=r and ii) when these three constants can assume arbitrary values. In both cases we will require that $r = r_1 = r_2 = r_3$ for tractability. Our cost function will be the sum of the cost of each version and the acceptance test and/or voter. We first examine the case where the system reliability R is fixed and the reliabilities r,V and B are computed to minimize the cost function. In the general case our nonlinear optimization problem becomes

$$\text{Minimize } C(r_1, r_2, r_3, B, V) \qquad \text{(O)}$$

subject to the constraint

$$S(r_1, r_2, r_3, B, V) = R.$$

Since reliabilities are probabilities, we have the addtional constraints that the r_i's, B, V and R must lie between zero and one. In section III we first treat the special case when

$$r_1 = r_2 = r_3 = B = V.$$

This reduces the above optimization problem to a straightforward root finding problem for functions of a single variable. It is more tractable than the general case and provides useful bounds.

III. Minimizing Cost subject to a Reliability Constraint

We will first treat the case where all exponents α are equal and all reliabilities are constrained to be equal. We also assume that the β values for the modules, β_r, the acceptance test, β_B, and voter, β_V, are 1.

N-version Programming

Since the model of N-version programming considered in [1] does not include an acceptance test and assumes a perfect voter with no cost ($V = 1$ and $\beta_V = 0$), we will treat it first. Since we are assuming that $r_1 = r_2 = r_3 = r$ and $\beta_1 = \beta_2 = \beta_3 = \beta_r = 1$, our cost function becomes $C(r) = 3(1-r)^{-\alpha}$ and the system reliability is

$$S_{nvp}(r) = 3r^2 - 2r^3 \qquad (4).$$

The function $S_{nvp}(r)$ is monotone on the interval [0,1] and hence the equation $S_{nvp}(r) - R = 0$ has a single real root, denoted by $r(R)$, in [0,1]. In this case the optimal cost is

$$C(r(R)) = 3/(1-r(R))^{\alpha}. \qquad (5)$$

Since we assume that $\beta_r = 1$, the cost of a system with a single or unit version with the same system reliability is $U(R) = 1/(1-R)^{\alpha}$. (5)

The right hand side of equation (5) is monotone increasing in R. In Table 1 we present its values for α = .5, 1, and 2 for R = .9,.95, .99, .999, 9999 and .99999.

		α=.5		α=1.		α=2.	
R	r(R)	C(R)	U(R)	C(R)	U(R)	C(R)	U(R)
.9	.804200	6.8	3.2	15.3	10	78.2	100
.95	.864650	8.2	4.5	22.1	20	163.8	400
.99	.941097	12.3	10	50.9	100	864.7	10000
.999	.981630	22.1	31.6	163.3	1000	8890	1000000
.9999	.994215	39.4	100	518.5	10000	89642	1 E 08
.99999	.998173	70.2	316.2	1642.0	100000	898760	1 E 10

Table 1: The cost of a 3 Version Programming system assuming a perfect voter and no cost. $\beta_r = 1$, $\beta_V = 0$

As one would expect, the value of α is critical in drawing conclusions when comparing the cost of a single version vs. a 3 version fault-tolerant system. When high system reliability is required it is more likely that a 3 version system will be more cost effective than a single version system in the case that the voter is perfect and has zero cost.

We now assume that the voter is neither perfect nor free. We let V be the reliability of the voter and $\beta_V = 1$. Our model for the reliability of a 3 version system where all the version reliabilities are equal becomes

$$S_{nvp}(V,r) = V(3r^2 - 2r^3). \qquad (6)$$

Assuming $\alpha_V = \alpha_r$ and V=r the reliability constraint becomes $R = 3r^3 - 2r^4$. If we graph the function $3r^3 - 2r^4$ for $0 \leq r \leq 1$ we find it is monotone and lies below the line y=r (see figure 1). The cost for this model is $C(r(R)) = (3+\beta_V)/(1-r(R))^\alpha$. In table 2 we assume that $\beta_V = 1$, hence $C(r(R)) = 4/(1-r(R))^\alpha$. The cost of a unit version is the same as that given in Table 1 and is omitted.

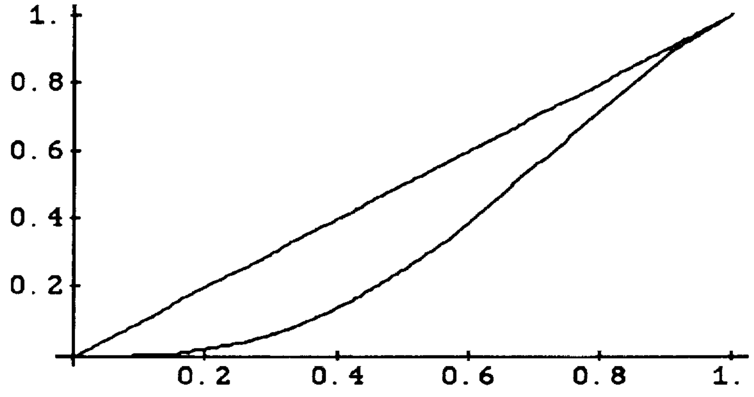


Figure 1. A graph of $S_{nvp}(r)$ with an imperfect voter and V=r

This gives the following table 2:

R	r(R)	α=.5	α=1.	α=2.
.9	.917647	13.9	48.6	589.8
.95	.955506	19	89.9	2020.5
.99	.990279	40.6	411.5	42329
.999	.999003	126.7	4012	4.0 E 6
.9999	.999900	400	40000	4 E 8
.99999	.999990	1264.9	4 E 5	4 E 10

Table 2: The cost of a 3 Version Programming system assuming a non-perfect voter. $\beta_r = \beta_V = 1$

Note that the imperfect voter causes r(R) to be larger for each R as compared to the perfect voter case. Larger r(R) implies a greater cost per version. It is clear that a simplex system with the same system reliability will now be less costly because $r(R) \approx R$ for R close to 1. Since we are paying (in terms of α) the same for the voter as for the versions it is never the case that a nonperfect voter 3MR software system will be more cost effective than a simplex system. This is a startling result. We will see that we will have similar results when we remove the constraint that $V = r$.

Recovery Block

If B=r then the system reliability becomes

$$S_{rb}(r) = 4r^6 - 8r^5 + 2r^4 + 2r^3 + r^2 \quad (7)$$

This sixth degree polynomial is also monotone in [0,1] which implies $S_{rb}(r) - R$ has a single real root in [0,1]. Figure 2 includes a graph of $S_{rb}(r)$. Again assuming that $\beta_B = \beta_r = 1$ and $\alpha_B = \alpha_r$ we have Table 3 which gives the values of $C(R) = 4/(1-r(R))^{\alpha}$.

R	r(R)	α=.5	α=1	α=2
.9	.790108	8.7	19.0	90.8
.95	.844921	10.2	25.8	166.3
.99	.922202	14.3	51.4	660.9
.999	.971875	23.8	142	5047.8
.9999	.990438	40.9	418.3	43748
.99999	.996886	71.7	1284.5	4.1 E 5

Table 3: The cost of Recovery Block, B=r, $a_a=a_a$

Comparing tables 1 and 3 it is clear that Recovery block is more cost effective than 3-version programming with a perfect voter and is more cost effective than a single version system for high reliability cases.

Consensus Recovery Block

The reliability of the consensus recovery block [2] is given by (3).

If V=1, i.e., the voter has reliability 1, and $\beta_V = 0$, then $S_{nvp}(V,r)$ is given by equation (1). If we also assume that B=r then $S_{rb}(r,B)$ is given by equation (3) and we have

$$S_{crb}(r)= 8r^9-28r^8+28r^7 + 2r^6 -12r^5 - r^4 +4r^2 \quad (8)$$

This function is monotone over [0,1]. Table 4 assumes that $\beta_B = 1$ and the voter has no cost. Hence $C(r(R)) = 4/(1-r(R))^\alpha$. We see that Consensus Recovery Block with a perfect voter is more cost effective than any of the previous systems as one would expect and the cost grows relatively slowly as R increases. For high reliability systems it will be signficantly cheaper that a single version system.

R	r(R)	a=.5	a=1	a=2
.9	.632687	6.6	10.9	29.6
.95	.695174	7.2	13.1	43
.99	.796570	8.9	19.7	96.7
.999	.882487	11.7	34	289.7
.9999	.931504	15.3	58.4	852.6
.99999	.960196	20	100.5	2524.7

Table 4: The Cost of Consensus Recovery Block with a perfect voter. $B = r$, $\beta_B = \beta_r = 1$, $\beta_V = 0$ and $\alpha_B = \alpha_r$.

If we use equation (2) for N version programming and equation (3) for recovery block then we have

$$S_{crb}(r) = 8r^{10} - 28r^9 + 28r^8 - 2r^7 - 11r^5 + 5r^3 + r^2 \quad (9)$$

This is also monotonic on [0,1] (see figre 2). Table 5 gives the values when all α's, r's and V and B are equal. If we assume $\beta_V = \beta_B = 1$ then the cost function is $C(R) = 5/(1-r(R))^\alpha$. This equivalent to assuming that the cost of the development of the voter and the acceptance test is the same as the cost of developing the modules which is usually not the case. Hence table 5 gives an upper bound on the costs.

R	r(R)	α=.5	α=1	α=2
.9	.698340	9.1	16.6	54.9
.95	.755889	10.1	20.5	83.9
.99	.847012	12.7	36.7	213.6

.999	.920723	17.8	63	795.6
.9999	.959660	24.9	123.9	3072.5
.99999	.980051	35.4	250.6	12563

Table 5: Consensus Recovery Block, V=B=r, $\alpha_V = \alpha_B = \alpha_r$.

Graphs of equations 7 and 9 are given in figure 2.

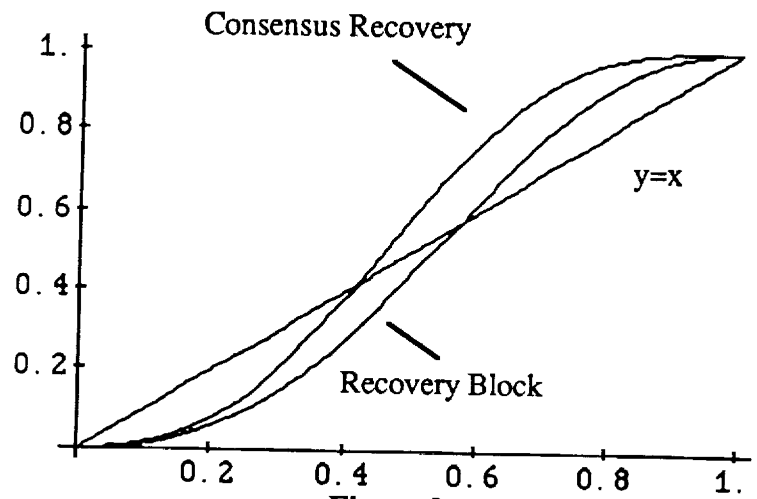


Figure 2: Reliability of the Recovery Block and Consensus Recovery Block assuming r = B = V

From tables 1 thru 5 it is clear that the most cost effective system in terms of total cost is consensus recovery block with a perfect voter followed by consensus recovery block with an imperfect voter. In general, consensus recovery block will be significantly less costly than either N-version programming, recovery block or a single version system.The least cost effective system is N version programming with an imperfect voter. It is even worse than a simplex system with the same reliability. Since we have assumed that all reliabilites are equal and all α's are equal these results provide an upper bound for the case that r, V and B are not required to have equal reliabilities. We will demonstrate this in the following section.

IV. Eliminating the equal reliability constraint r = V = B

In this section we treat the case when r, B and V can have different values in the optimization problem (O). We will minimize the cost subject to the constraint that the system reliability must be met.

We have chosen to use the constrained optimization technique of Lagrange multipliers [5]. Applying this technique yields the following optimization problem: Let λ be a 'Lagrange multiplier', $C(x_1,x_2,\ldots,x_n)$ the objective function to be optimized and let $G(x_1, x_2, \ldots x_n) = K$ be the constraint. We form the function

$$u = C(x_1,x_2,\ldots,x_n) + \lambda\, G(x_1,x_2,\ldots,x_n)$$

A solution $(x_1, x_2, \ldots, x_n, \lambda)$ to the following set of nonlinear equations is an optimal solution to the original optimization problem:

$$\begin{gathered}\partial u/\partial x_1 = 0\,,\\ \partial u/\partial x_2 = 0\,,\\ \vdots\\ \partial u/\partial x_n = 0\\ G(x_1, x_2, \ldots, x_n) = K\end{gathered}$$

We have applied unconstrained Newton's method for several variables [6] successfully for this problem. The requisite partial derivatives were calculated symbolically using Mathematica [4].

Some discusssion of the numerical properties of our iterative technique is in order. Newton's method does not guarantee convergence for arbitrary starting values. Futhermore, convergence can occur at a point for which r, V or B lies outside the allowable range, i.e., these values must be probabilities and lie in the interval [0,1]. Hence starting values are critical. The authors used a Pascal program called MINCOST which runs on an IBM mainframe under VM. All calculations were in double precision which is approximately 14 decimal digits. The program allows the user to choose initial values for r, B, V and λ. Newton's method uses a linearization of the nonlinear equations and solves the linearized version to calculate correction values to the current estimate of the solution. Once the correction values are sufficiently small or the number k of allowable iterations is exceeded the iteration is halted. If convergence has taken place and the values of r, B or V lie outside the allowable range or if the number of iterations k has been exceeded then a search for a better starting value begins. This is accomplished by adding and subtracting a change value δ to each of r,V and B until convergence in range takes place. If no convergence occurs for a given δ, then 2δ is tried and the search for convergence in range begins again. The process continues until convergence is achieved or a reliability lies outside [0,1]. If the system is used to find optimal values for several different R's then arranging these values in ascending order, $R_1 < R_2 < \ldots < R_m$, and using the solutions for R_i as starting values for the optimization problem for R_{i+1} usually gives good results, especially if the R_i's are 'close'.

N version Programming

Recall that if $r_1 = r_2 = r_3$ equation (6) gives the reliability

$$S_{nvp}(r,V) = V(2r^3 - 3r^2)$$

Recovery Block

If B not equal r then $S_{rb}(r,B)$ becomes the generalized cubic polynomial

$$S_{rb}(r,B) = 4B^3r^3 - 4B^3r^2 + B^3r - 4B^2r^3 + B^2r + Br^3 + Br^2 + Br . \qquad (10)$$

We note that the surface $S_{rb}(r,B)$ is symmetric in r and B.

Consensus Recovery Block

$S_{crb}(r,B,V)$ is given by equations 6 and 10 in terms of S_{nvp} and S_{rb} and equals

$$\begin{aligned} &r(B + B^2 + B^3 + Br - 4B^3r + Br^2 - 4B^2r^2 + 4B^3r^2 + 3rV - 2r^2V - 3Br^2V - \\ &\quad 3B^2r^2V - 3B^3r^2V - Br^3V + 2B^2r^3V + 14B^3r^3V - Br^4V + \\ &\quad 12B^2r^4V - 20B^3r^4V + 2Br^5V - 8B^2r^5V + 8B^3r^5V) \qquad (11) \end{aligned}$$

Numerical Results

The following discussion is based on the numerical results in Tables 1 through 88 in the Appendix. We first observe that the single variable case discussed previously is a relatively tight upper bound for the case when all β values are equal. Hence for quick approximations, assuming that all reliabilities, β's and α's are equal gives good results and is considerably more tractable.

Figures 3,4, and 5 are plots of the version, voter and acceptance reliabilities from R = .98 to R = .99999 in the optimal solution for the case that the β's are 1 and the α's are equal and set to 1. The cases for $\alpha = .5$ and $\alpha = 2$ are similar. We note that in figure 3, the voter must be considerably more reliable than the versions and that a simplex system is *always* less costly by a factor of 2 to 3. As shown in figure 4 the acceptance test must also be more reliable than the versions but by at most .05. The recovery block is always less costly than a simplex system in this range with the difference in costs increasing as higher system reliability is required. The consensus recovery block is also always less costly than a simplex system. Figure 5 shows the acceptance test must be more reliable than either the voter or the versions and there is a crossover point where the voter must be more reliable than the versions. The cost ratio between consensus recovery block and recovery block is approximately .5 to .6. For $\alpha = .5$, a simplex system was less costly than either recovery block or consensus recovery block until system reliability was above .99. See Tables 3 and 5. This was not true for $\alpha = 1$ or $\alpha = 2$. This implies that as α increases, both recovery block and consensus recovery block will be relatively less costly for high system reliability. Figure 6 shows the optimal cost vs. system reliability of each of the fault-tolerant techniques considered.

Since the objective function is linear in the contants β_i, for a given fault-tolerant technique holding the α's fixed will result in identical optimal values for r, V and B for a given system reliablity R if we multiply all β's in the objective function by the same constant k. Hence, the problem is interesting only when the values of the β's are changed relative to each other for given values of the α's. To avoid getting lost in a morass of data we have partitioned the problem into two cases. In both cases the β values for the versions are set to 1 and we calculate the optimal system costs for the cases R = .9, .95, .99, .999, .9999, and .99999. These results are shown in table 1 through 88 in the Appendix.

In case 1 both the β values for the acceptance test and the voter are equal and vary from .1 to 1. in steps of .1. We plot these results for the case R=.99999 in figures 7,8 and 9. We discuss the results for this value of R below.

We see in figure 7 that for $\alpha = .5$ that nvp is less costly than a single version until β_V is approximately .6 or larger. For this case the reliability of the versions increase from .998 to .9993 in nvp, from .983 to .992 in rb and from .935 to .964 in crb. The reliability of the voter remains at .99999 in nvp and decreases from .993 to .984 in crb. The reliability of the acceptance test decreases in rb from .9997 to .9988 and in crb it decreases from .996 to .988. The cost of nvp increases from 128 to 455, rb increases from 29 to 63 and crb increases from 15 to 33. The cost of a single version is 316.

In figure 8, for $\alpha = 1$, nvp becomes more costly for β_V approximately .9. The reliability of the versions increase from .9992 to .9996 in nvp, from .988 to .994 in rb and from .946 to .970 in crb. Again, the reliability of the voter remains at .99999 in nvp while it decreases from .990 to .983 in crb. The reliability of the acceptance test decreases in rb from .9993 to .9981 and in crb it decreases from .993 to .985. The cost of nvp increases from 16006 to 112377, rb increases from 402 to 1119 and crb increases from 82 to 231. The cost of a single version is 100,000. The differences in the reliabilities remain remain relatively small between the two values of $\alpha = .5$ and 1.

In the case that $\alpha = 2$, shown in figure 9, nvp approaches the cost of a single version system as β_V -> 1. For this case the reliability of the versions increase from .9997 to .9998 in nvp, from .993 to .994 in rb and from .959 to .975 in crb. The reliability of the voter remains at .99999 in nvp and decreases from .988 to .982 in crb. The reliability of the acceptance test decreases in rb from .9986 to .9976 and in crb it decreases from .989 to .983. The cost of nvp increases from 1,086,190,062 to 10,269,661,722, rb increases from 117,303 to 358,182 and crb increases from 3,352 to 11,449. The cost of a single version is 10^{10}.

It is clear that α has a more significant affect on cost than either β_B or β_V. In all cases rb and crb are much less costly than nvp or a single version. Both rb and crb are approximately loglinear for larger values of β_V and β_B less than 1.

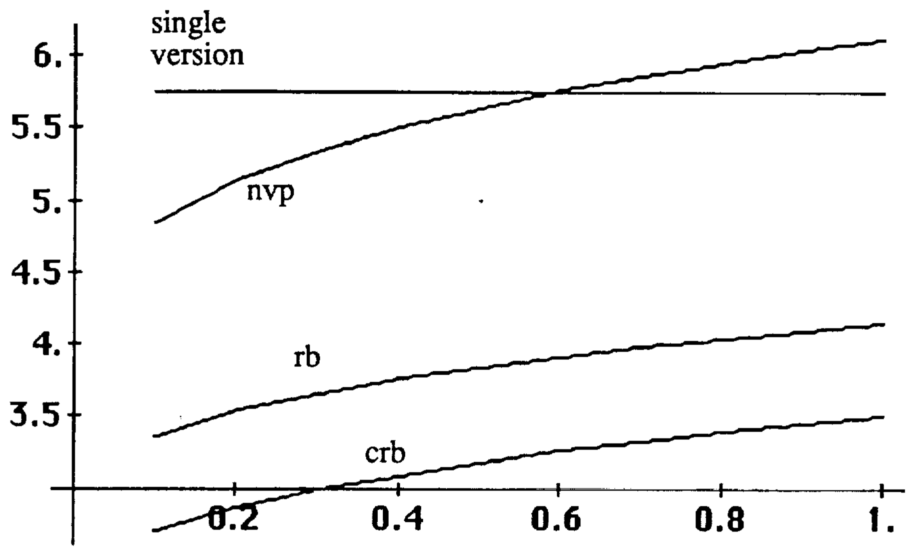


Figure 7. Log (costs) for R= .99999, $\alpha = .5$, $\beta_V = \beta_B = .1(.1)1$

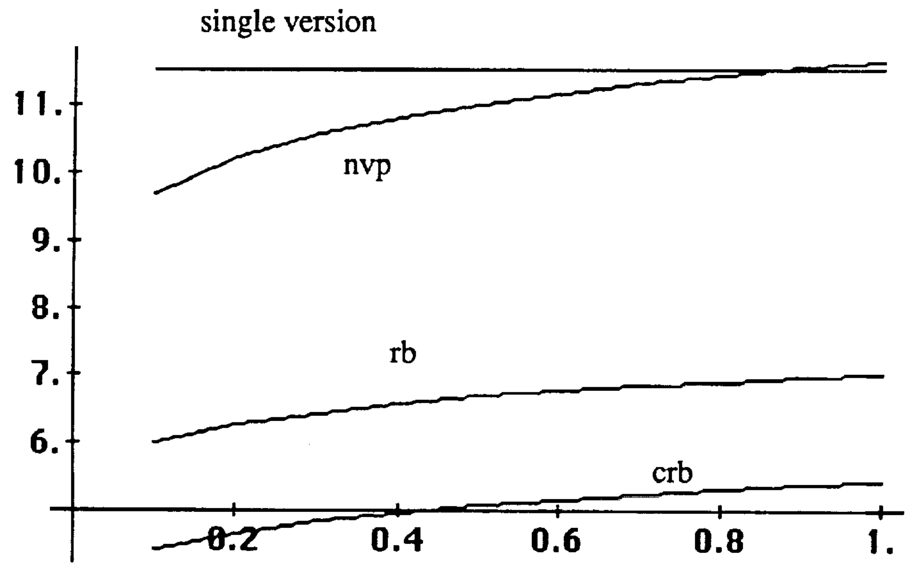


Figure 8. Log(costs) for R=.99999, $\alpha = 1$ $\beta_V = \beta_B = .1(.1)1$

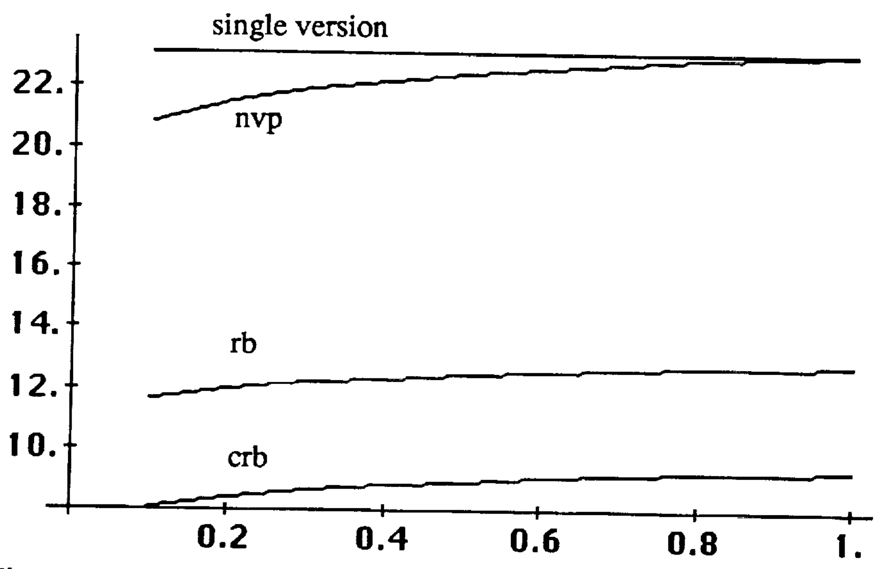


Figure 9. Log(costs) for R=.99999, $\alpha = 2\beta_V = \beta_B = .1(.1)1$

In case 2 since the consensus recovery block requires an acceptance test and a voter, to gain some intuition on the relationships between β_V and β_B we vary them from .25 to 1 in steps of .25. As in case 1 the β's for the versions are 1. In figure 10 we show the cost as a two variable function of β_V and β_B for the case $\alpha = 1$ and R = .99999. The cases for $\alpha = .5$ and 2 are similar.

We see that the function is monotone in both variables. The cost is slightly more affected by an increase in β_B.

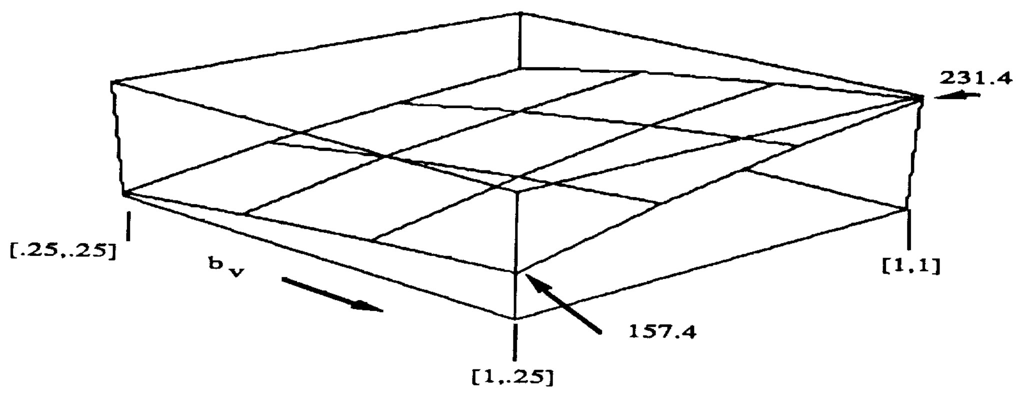


Figure 10. Plot of cost of Consensus Recovery Block as a function of β_V and β_B for the case $\alpha = 1$. R = .99999; β_B and $\beta_V = .25(.25)1$.

The results were relatively consistent among all cases considered. We present our results in tables in the Appendix where the optimal values for r, B, V, C and the values of the β's , and α's are given as a function of the system reliability R.

V. Summary and Conclusions

The above results have shown that in the case that failures are independent, Consensus Recovery Block and Recovery Block are the only cost justifiable fault-tolerant techniques to be considered. Unless the voter is perfect, N-Version Programming does not compete cost-wise with the other two methods. However, the hybrid method Consensus Recovery Block which contains both voting and recovery block can provide considerable reduction in cost for a given system reliability over the other techniques.

We are currently attempting to relax the condition that all version reliabilities are equal and that failures are independent. We intend to move to 5 and 7 version systems to determine how costs are related. We are also investigating another hybrid system which is called acceptance voting where an acceptance test is used after each version completes. Only those outputs which have been determined correct are then passed to a voter. We will report on this and other items in a later paper.

Bibliography

1. R. K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", IEEE FTCS 14,1984, pp. 102-107
2. R. K. Scott , J.W. Gault and D. F. McAllister,"The Consensus Recovery Block ", Proc. Tot. Sys. Rel. Symp.,Dec. 1983, pp. 74-85
3. R. K. Scott, J.W. Gault and D.F. McAllister, "Modeling Fault-Tolerant Software Reliability", Third Symposium on Reliability in Distributed Software and Database Systems, Oct. 1983, pp. 15-27
4. Stephen Wolfram, **Mathematica**, Addison-Wesley, 1988
5. Anlgus E. Taylor and W. Robert Mann, **Advanced Calculus**, Second Edition, John Wiley and Sons, New York, N.Y. 1972, pp. 197-198
6. Terry E. Shoup, **Numerical methods for the Personal Computer**, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1983, pp. 64-69

3. Analysis of Faults Detected in a Large-Scale Multi-Version Software Development Experiment*

Mladen A. Vouk,
David F. McAllister
North Carolina State University

Alper K. Caglayan
James L. Walker, Jr.
Charles River Analytics, Inc.

David E. Eckhardt
NASA Langley Research Center

John J.P. Kelly
University of California, Santa Barbara

John Knight
University of Virginia

Abstract

Twenty programs were built to the same specification of an inertial navigation problem. The programs were then subjected to a three phase testing and debugging process: an acceptance test, a certification test, and an operational test. Less than 20% of the faults discovered during the certification and operational testing were non-unique, i.e. the same or very similar faults would be found in more than one program. However, some of these "common" faults spanned as many as half of the versions. Faults discovered during the certification testing were due to specification errors and ambiguities, inadequate programmer background knowledge, insufficient programming experience, incomplete analysis, and insufficient acceptance testing. Faults discovered during the operational testing were of a more subtle nature, and were mostly due to various programmer knowledge defects and incomplete analysis errors. Techniques that may be used to avoid the observed fault types are discussed.

1. Introduction

Most common fault-tolerant software mechanisms are based on redundancy [e.g. 1, 2]. For acceptable results these techniques require that the failures of redundant software versions are mutually independent, or at least that the positive inter-version failure correlation is low [3]. Experiments have shown that failure dependence among independently built programs intended to be functionally equivalent may not be negligible in the context of the current software development and testing techniques [e.g. 4, 5, 6, 7]. Therefore, it is important to understand the nature of the faults that may cause dependent failures in such programs. NASA LaRC sponsored a multi-university[6] experiment to develop a set of high reliability aerospace application programs which would be used to study multi-version software. Twenty programs were built and tested by two-person programming teams. After that the programs went through a three phase testing and debugging process. Following minimal acceptance testing, all programs were subjected to extensive certification testing and debugging, and then operational testing. During certification testing over 1100 designed and random test cases were used, and the detected faults were corrected. During the operational testing we executed over 900,000 test cases aimed at identification of the faults that may not have been captured in the previous phases.

*This research was supported in part by NASA grants NAG-1-511, NAG-1-667,NAG-1-744, and NAG-1-782, and contracts NAS-1-17705 and NAS-1-17964.

[6]Participating universities: North Carolina State University (NCSU), University of Illinois Urbana Champagne (UIUC), University of California Los Angeles (UCLA), University of California Santa Barbara (UCSB), University of Virginia (UVA). Participants from industry were: Research Triangle Institute (RTI), Charles River Analytics (CRA). Phase 1 of the experiment was coordinated by RTI with assistance from CRA. Phases 2 and 3 of the experiment was coordinated by CRA.

In this paper we examine the character of the similar faults found in the versions generated during the experiment. In the following section we briefly describe the experiment in which the software components were produced. In section 3 we describe the profile of the detected faults, and discuss some possible avoidance strategies. The summary is in section 4.

2. Experiment

The programs solve a problem in inertial navigation. The requirement was to interpret and analyze accelerometer sensor signals received from a redundant strapped down inertial measurement unit (RSDIMU). The code was written in Pascal, and developed and tested in a UNIX environment on VAX hardware. Additional information about the experiment can be found in [7, 8].

At the end of the original development phase all programs were acceptance tested with a set of 75 test cases. Because the development had revealed a number of specification related problems the base specification document, before being used in the certification testing, was augmented with the clarifications and the amendments sent to the RSDIMU programmers during the original development. For certification testing and debugging the programs were re-assigned among twenty new programmers located at NCSU, UCSB and UVA.

The certification test suite consisted of 801 extremal and special value test cases (see section 3.2) and 400 random test cases. The test cases were constructed using the "black-box" approach, and with the intention of providing maximal functional coverage of the requirements specifications. The random data served as a check on the the designed cases. Although some faults related to numerical instabilities, round-off errors and program "memory" were uncovered the test suit was not designed to specifically search for such faults. During certification testing the correctness of the answers was adjudicated using a "golden" program. This program has been very extensively tested and inspected on its own. The operational testing of the golden program supports our belief that it contains no errors. Use of back-to-back testing as an alternative failure identification method is discussed in section 3.4.

In making comparisons with the answers from the "golden" program we used tolerances compatible with the accuracy of the input data. Eleven output variables (59 individual values) were checked for each input. A difference was signaled whenever any one of these values differed from the corresponding "golden" value. When differences were discovered the programmers were requested to investigate the problem, correct it, and submit a software change and correction report detailing the fault description, its symptoms and any changes. Regression testing with the full test data set was applied to all re-submitted software.

In the analysis phase of the experiment, the delivered code was inspected manually, and possibly re-run, in order to precisely identify software faults, and establish the reasons for non-execution of certain parts of the code. The cumulative execution coverage of the code by the test cases was determined through code instrumentation and execution tracing. A more detailed description of the certification testing effort can be found in [9, 10].

In the operational testing phase the programs were subjected to an extensive test, totalling over 900,000 input cases, in order to determine software reliability. The correctness of the submitted software was asserted using a test which had prior knowledge of the aircraft simulation, and which implemented differential equations for translational dynamics, rotational kinematics, and an aircraft guidance law. The versions were tested using the test harness with several sets of simulated flight data. Fifteen output variables were monitored in this phase. If a failure was detected, the test harness automatically recorded all inputs to the harness and all program outputs. The fault causing a failure was found through isolated execution of the version code with the input set that resulted in the failure, and dynamic (using a debugger) and static inspections of the code. Additional information about the operational testing process and results can be found in [8, 9].

3. Faults

3.1 Fault Classification

We divide the faults into two principal classes: the specification related faults (S), and the design and implementation related faults (D):

- **Specification related faults** are those that can be directly traced to problems (errors) within the original, English language, requirements specification document. They include semantic ambiguities (that resulted in multiple interpretations of parts of the document), errors in the specification content, typographical errors, and omissions. Most of these errors led to changes in the requirements specification document prior to certification testing.
- **Design and implementation faults** are non-specification related faults that could be traced to errors made by individual teams or programmers during software design, coding or program testing. An error made by a programmer may be due to a misunderstanding of some concept presented in the specifications, or it may be due to a lack of knowledge, or it may be a question of an overlooked special case, etc.

We also classify the faults by their causes. This is discussed in section 3.4.

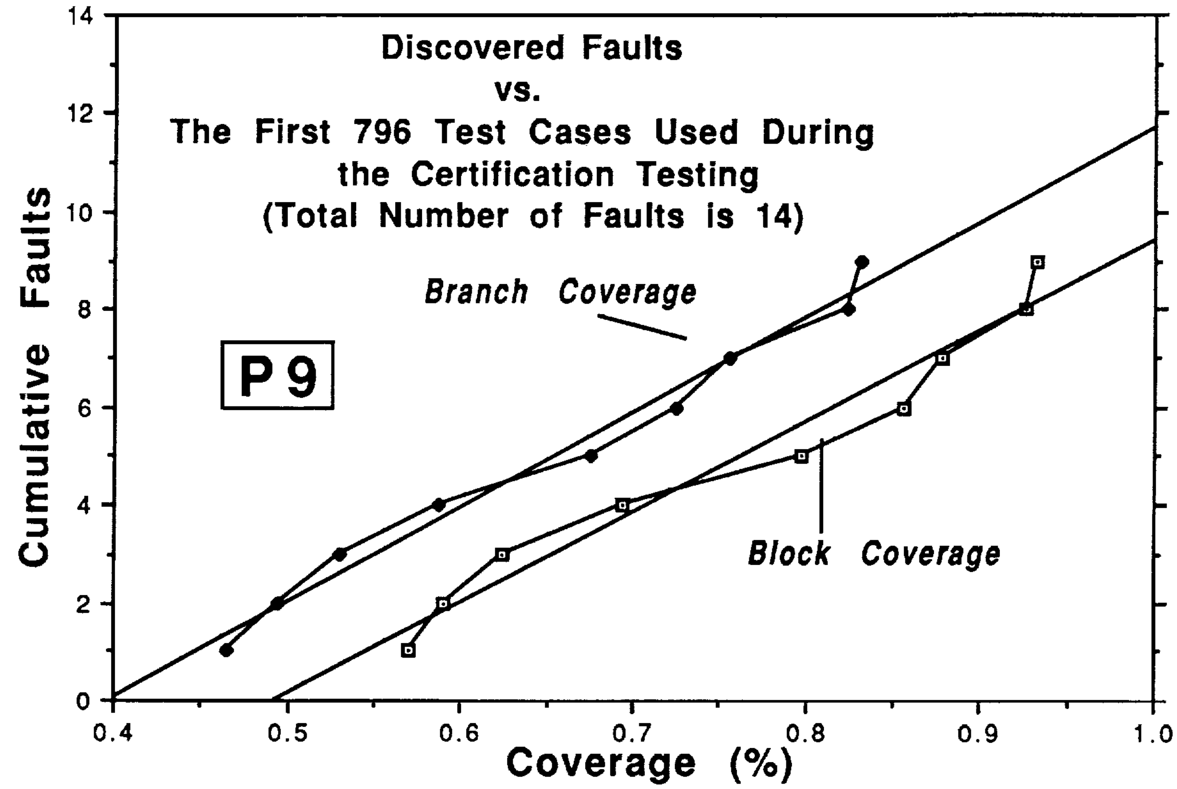


Figure 1 Coverage based fault detection growth.

3.2 Certification and Operational Testing

During certification testing we detected a number of faults of varying prevalence and severity. Some of the faults were found to result in highly correlated coincident failures of several programs. In addition, through code and output inspections, one fault was detected in the testing harness and one fault was found in the "golden" program. These faults were corrected, and the appropriate patches and an additional 5 test cases were distributed to all sites before the testing was concluded. These additional test cases were aimed at exercising the functionality that was missed in the golden code. The rationale for adding more extremal and special value test cases to the 796 already in the set was that the fault was discovered during the system testing and therefore it should be removed from all versions, including the "golden" one, during that process.

Fault detection efficiency of the first 796 designed (black box) test cases used for certification testing is illustrated in Figure 1 for program P9. The figure shows the cumulative number of faults detected through structural coverage provided by the cases.

Table 1 lists the prevalence of the faults by the phase in which they were detected, and the phase in which they were committed (specifications, implementation). During certification testing we detected several groups of specification related faults, and over 60 distinct design and implementation faults. A change in the specification requirements regarding a call to a voter function (itself not an integral part of the inertial navigation problem) required changes in all 20 versions at the beginning of the certification phase. This change (C4) is not listed in Table 1 but faults associated with this change are in the table.

Table 1. The total number of faults detected in the programs during certification testing, and during operational testing.

	Certification Testing		Operational Testing
Prog.	Spec Faults*	Design Faults	Design Faults
P1	2	2	-
P2	2	5	1
P3	2	6	1
P4	2	8	2
P5	2	8	1
P6	1	5	-
P7	2	6	3
P8	-	6	3
P9	2	8	4
P10	-	2	1
P11	1	3	3
P12	$\geq$2**	$\geq$4**	-
P13	1	8	-
P14	2	5	1
P15	1	5	1
P16	2	6	-
P17	1	4	2
P18	1	6	3
P19	1	6	-
P20	-	6	-

* C4 is not counted.

** There were four major overhauls of the P12 code because the original was very defective. In the process, at least two specification and four design and implementation faults were removed.

The certification testing placed more functional demands on the software than the original acceptance test and, as a result, many software errors were corrected. However, not all errors present in the original versions were eliminated by the certification testing and debugging. During the operational testing we did not detect new specifications related faults, but many of the faults we did discover were of a very subtle nature and could only be encountered through extensive dynamic testing. We know of at least 15 distinct faults that propagated to the operational testing phase, eleven of which are faults not recorded during certification testing in any of the programs. It is interesting to note that in two programs a partial correction of a fault resulted in a more serious fault through overwriting of a primary input variable (fault D8.1 of Table 2.1 became D8.4 in Table 2.2).

Table 2.1 Similar faults detected in certification testing.

Fault	Brief Description
C1	Misalignment correction problems (specification misinterpretation/ambiguity)
S1.1	Assumes misalignment angles are zero
D1.1	Combined values form two distinct frames of reference
D1.2	Corrected misalignment twice, the second correction introduces an error
D1.3	Used NORMFACE in the wrong frame of reference
C2 (S2)	LINOFFSET and other values not computed for failed sensor on input (specification ambiguity, implementation faults)
C3	Sensor failure detection and isolation (FDI) problems (specification ambiguity, implementation faults)
S3.1	Performs FDI for violation of 2 or 3 edge relations
S3.2	Performs FDI for violation of 1, 2, or 3 edge relations
D3.1	Test threshold for edge vector test is miscalculated due to an uninitialized variable
D3.2	"Fails the system" when 1 or 2 edge relations are violated
D3.3	"Fails the system" when 1 edge relation is violated
D3.4	Performs FDI for violation of 1 or two edges (erratic, initialization problem)
D3.5	Erratic FDI due to uninitialized variable
D3.6	FDI works only if precisely tree edge relations with a common face are violated (fatal run-time failure possible)
D3.7	Always "fails the system" due to uninitialized variable
D4	Display round off error
D4.1	Fatal run-time failures (missing labels, out-of-range checks)
D4.2	Incorrect rounding of displayed acceleration value
D5	Division by zero on all failed input sensors (fatal run-time failure in all cases).
D6	Incorrect conversion factor for conversion of ft/s/s to m/s/s
D7	Votelinout procedure call placement fault and/or error in using returned values (precipitated by C4)
D8	Input from sensor **not** properly masked to 12 bits (e.g. modulo 4096 missing)
D8.1	Input arrays RAWLIN and OFFRAW not masked
D8.2	Only RAWLIN not masked
D8.3	Only OFFRAW not masked
C4	missing Votelinout voter call (specification **change/addition**)

3.3 Similar faults

In this paper we limit the discussion to faults that occur in more than one version, and to faults which have an increased potential to result in a coincidental and possibly identical-and-wrong (IAW) response from two or more versions. These faults are sometimes called similar faults. Similar faults may appear in several versions, they may affect the same functions and problem variables, and identical or similar parts of the code, or they may be unique to a program. What they have in common is that the failures they produce tend to correlate among versions. They fail coincidentally because their fail sets (i.e. input states which result in a failure) overlap. The failures are correlated because this overlap differs significantly from that expected by chance. Usually, the probability that the correlated versions fail coincidentally is higher than would be expected from versions that fail independently, but negative correlation is possible. A similar fault may manifest as a response different from that of any other version (for the same input data), or it may manifest as a IAW response from two or more versions.

The similar faults we have observed in certification testing are described in Table 2.1. The similar faults detected in operational testing are described in Table 2.2. The distribution of similar faults across the programs is shown in Table 3.1 after acceptance testing, and in Table 3.2 after certification testing.

The specification related faults are a direct result of ambiguities, inconsistencies, or lack of clarity in the requirements document. However, some design and implementation faults are also very closely related to functions affected by the specification problems. In fact, we feel that a considerable number of the similar faults were influenced by the communication problems between the specification writers and the programmers. We have divided these faults by functionality they affect into four groups denoted C1 through C4. Some of the faults in these groups can be clearly related to a specification problem, some are primarily implementation related, and some are related to a blend of the specification and design/implementation problems. All produce correlated failures.

Table 2.2 Similar faults detected in operational testing.

Fault	**Brief Description**
C1	Misalignment correction problems
D1.4	Implemented sensor failure/isolation in the wrong frame of reference
D1.5	Combined two different frames of reference in sensor failure/isolation algorithm
C3	Sensor failure detection and isolation (FDI) problems
D3.5	Erratic FDI due to uninitialized variable
D3.8*	Implemented sensor failure/isolation with incorrect unit vector
D3.9*	Combination of unlike values in sensor failure/isolation algorithms
D3.10	Incorrect computation of standard deviation in sensor failure/isolation algorithm
D3.11	Three out-of-tolerance edge relations were assumed to have a face in common with all violated relations
D3.12	Four violated edge relations were not processed
D6	Slightly incorrect factor for conversion of ft/s/s to m/s/s
D8	Input from sensor **not** properly masked to 12 bits (e.g. modulo 4096 missing)
D8.2	Only input array RAWLIN not masked
D8.4	OFFRAW masked but overwritten

(*) D3.8 and 3.9 are conceptually very similar and can be treated as a single conceptual fault

Two important groups are C1 and C3. The C3 group concerns the most important function of RSDIMU – failure detection and isolation. All but one version suffered from a problem in that area. Only two of the fourteen C3 faults are clearly ambiguity related, while the rest of them can be classified as implementation faults precipitated by incomplete specification analysis, application knowledge problems, etc. Five of the 14 were detected for the first time during operational testing. What for the programmers appears to have been the next most problematic portion of the specification – the process of translation of quantities between an aligned sensor frame and an unaligned sensor frame – manifests as the C1 group of faults. Only one of the six C1 faults, the S1.1 fault, is clearly a specification (content) error. The rest are more implementation oriented. Two were detected during operational testing. A specification related problem that is only important if staged-voting[7] is used is C2 (S2). Fifteen of the 20 versions had this problem. It can be traced to lack of clarity, and possibly an ambiguity, in the specifications and lack of programmer knowledge of the application. It is also interesting that a number of programmers did not implement the C4 change correctly on the first try (i.e. put a new voter routine in the proper place, or make proper use of the values it returns) which resulted in the D7 fault.

Faults which may be particularly dangerous in practice are those that produce coincident version failures with IAW answers of such span that a voter is confused or mislead into choosing an incorrect answer as the correct one. Most of the similar faults listed in Tables 3.1 and 3.2 are capable of producing IAW failures of two or more versions. The composition of the program tuples that fail coincidentally and yield IAW answers, and the size of these tuples, vary with the input data, the number and the type of monitored variables, and the fault mix present in the code.

Table 3.1 Distribution of similar faults after acceptance testing.

Fault	Program(s)	Fault Span
C1	(P2. P3. P4. P5. P7.P10.P14. P17. P18. P20)	
S1.1	P7	1
D1.1	P2.P3.P4.P5.P10	5
D1.2	P18	1
D1.3	P20	1
D1.4	P8. P14. P18	3
D1.5	P17	1
C2 (S2)	P1.P2.P3.P4.P5.P6.P9.P11.P12.P13.P14.P15.P16.P18.P19	(15)
C3	(P1 - P19)	
S3.1	P2. P3. P5. P9. P12. P14. P16	7
S3.2	P1. P17	2
D3.1	P15	1
D3.2	P8	1
D3.3	P2. P5. P12. P14. P16	5
D3.4	P6	1
D3.5	P7	1
D3.6	P13	1
D3.7	P19	1
D3.8	P4	1
D3.9	P10	1
D3.10	P11	1
D3.11	P8. P9. P18	3
D3.12	P8. P9. P11. P15. P18	5
D4	(P2. P3. P4. P5.P7.P8.P9.P12. P13.P15.P17.P18.P20)	
D4.1	P3.P6.P7	3
D4.2	P2.P4.P5.P9. P12.P13.P15.P17.P18.P20	10
D5	P3. P4. P5.P7. P13.P15.P20.	7
D6	P2. P4. P9	3
D7	P2. P4. P5. P6 .P8. P9. P10. P14. P18	9
D8	(P1. P2.P3.P5.P7.P8.P9.P11. P16. P17. P19)	
D8.1	P1.P3.P5.P8.P9.P11.P17.P19	8
D8.2	P7	1
D8.3	P2. P16	2

For example, using certification test suite on versions that have completed acceptance testing revealed that a three-element real-valued vector which returns a very important (best acceleration)

[7]Staged voting is an approach where the code of each component is divided up into several stages. After each stage, voting takes place and the answer that is selected as the correct one is passed on to the next stage of all the components.

estimate had maximum IAW span of six in four different groups of versions (P2,P3,P4,P5,P9,P10; P2,P4,P5,P14,P15,P17; P2,P5,P8,P14,P15,P17; P2,P3,P5,P8,P11,P17). Over the same test set two other important variables, the eight-element binary-valued sensor failure vector and the binary system status variable, had maximum IAW answer spans of 6 and 8 respectively. The first was identically wrong for one group of 6 versions (P1,P2,P8,P11,P13,P18), while the other for one group of eight versions (P1,P2,P4,P5,P8,P14,P16,P17). From Table 3.1 we see that these IAW responses cannot be attributed to a single similar fault but to a combined effect of several faults. It is obvious the IAW answer effect is the result of a complex interaction among input values and the defects in the code, and that it may be difficult to predict the IAW properties of a group of faults solely from their physical characteristics and static program analysis.

A comparison of Tables 3.1 and 3.2 shows that although certification testing did not eliminate all similar faults, it has eliminated a considerable number of them. It also shows that the span of the remaining faults does not exceed five versions.

Table 3.2 Distribution of similar faults after certification testing.

Fault	Program(s)	Fault Span
C1	(P8,P14, P17, P18)	
D1.4	P8, P14, P18	3
D1.5	P17	1
C3	(P4,P7,P8,P9,P10,P11,P15,P18)	
D3.5	P7	1
D3.8	P4	1
D3.9	P10	1
D3.10	P11	1
D3.11	P8, P9, P18	3
D3.12	P8, P9, P11, P15, P18	5
D6	P2, P9	2
D8.4	P3,P17	2

Some of the similar faults are good examples of inadequate analysis, design or unit testing. The D4 fault affects a function clearly described in the specification document – rounding of displayed digits. Fault D5 induces fatal execution-time failures in seven programs. In six, it is the result of an attempt to divide by zero in the rare case when all sensors are assumed to have failed on input. Both faults are straightforward examples of functionalities requiring a test case for extremal values, yet many teams did not think of checking their code for proper functioning under these circumstances. Several other instances of fatal run-time failures were observed for similar faults (e.g. component P13 for D3.6, and components P3, P7 and P8 for D4.1), as well as for dissimilar faults. The D8 faults are another example of the situation where identical errors were made by many different programmers because explicit specification instructions on the masking of input data to only 12 bits were either not correctly understood or were not correctly implemented.

The D6 fault is related to (a slightly) incorrect conversion of the acceleration due to gravity from feet/sec^2 to meters/sec^2. The problem may have been alleviated in part had the specifications directly referenced the publication containing standard conversion data. The D7 is not critical for the operation of the code in the absence of staged-voting, but by its presence it reflects on the potential reliability of staged-voting approach.

An interesting group of faults is related to variable initialization lacks (e.g. S2, D3.1, D3.4, D3.5). For example, when all program variables are initially set to zero (which is often done by interpreters and compilers) most of the programs that have the S2 return zero as the acceleration estimates due to erroneous declaration of the system failure. However, when specially selected initialization values were used, one program returned zero because it overrode the system initialization through its own, unrequested, initialization code. On the other hand, D3.1, D3.4 and D3.5 are the result of omissions to initialize certain variables.

3.4 Fault Avoidance

In general, there is no single dominant cause for the observed software failures and faults. The analyzed software faults were due to specification errors, inadequate programmer background knowledge in the application domain, conceptually hard problems, insufficient programming experience, inadequate unit testing during development, insufficient acceptance testing, inadequate software development tools, lack of programming language support for certain programming constructs, etc. This is illustrated in Tables 4.1 and 4.2 where we have grouped the faults by their causes.

The four classes that we use for this purpose are: "Communication Errors" which includes specification ambiguity, specification omissions, specification clarity problems, confusion stemming from responses received by programmers to their questions about certain the specification items, and any other errors which could be attributed to problems with communication of requirements or other technical and process issues to programmers, "Defective Knowledge Errors" which includes lack of knowledge in the application area, lack of knowledge in the area of the programming language, inadequate design and testing methodology, errors in managing the software process, etc, "Incomplete Analysis Errors" which includes inadequate effort spent on conceptually difficult problems, misunderstanding of correct specifications, lack of understanding of all possible conditions that can occur at a given point in program, etc., and "Transcription Errors" which are typos.

Table 4.1 Causes of similar faults discovered during certification testing

Cause	Smilar Faults
Communication Error	S1.1, S2, S3.1, S3.2, D7, (D1.1, D1.2, D1.3, D6, D8)
Defective Knowledge Error	D3.1, D3.4, D3.5, D3.7, (D1.1, D1.2, D1.3, D8)
Incomplete Analysis Error	D1.1, D1.2, D1.3, D4, D3.2, D3.3, D3.6, D5, D6, D8,
Transcription Error	-

We have cross-listed some of the faults which we feel originate from a strong admixture of causes. The parenthesized lists give the faults for which the cause is a possible "secondary" influence. If we consider only the primary cause then we see that after the acceptance testing (both tables combined) the most numerous are the incomplete analysis (11) and the defective knowledge faults (10). However, if we also consider the possible secondary causes, we notice that an approximately equal number of faults suffer from the communication (11), the defective knowledge (14), and the incomplete analysis problems (14). The transcription errors are conspicuous by their absence, although some dissimilar faults may have an origin, at least in part, in transcription errors (e.g. an error in P19 where the implemented equation had incorrect sign in one place).

Table 4.2 Causes of similar faults discovered during operational testing

Cause	Smilar Faults
Communication Error	(D1.4)
Defective Knowledge Error	D1.4, D3.5, D3.8, D3.9, D3.10, D3.11, D3.12, (D8)
Incomplete Analysis Error	D1.5, D6, D8, (D3.8, D3.11, D3.12)
Transcription Error	-

Particularly problematic are fault groups C1 and C3. If a formal specification language were used instead of English, certain gross mistakes (e.g. FDI ambiguity that results in S3.1 and S3.2 faults) detected before acceptance testing would have been possibly avoided at the specification writing stage. However, it is safe to state that a large portion, if not most, of the subtle errors in the versions (e.g. C3 group from Tables 2.2 and 3.2) would still be present after the certification testing.

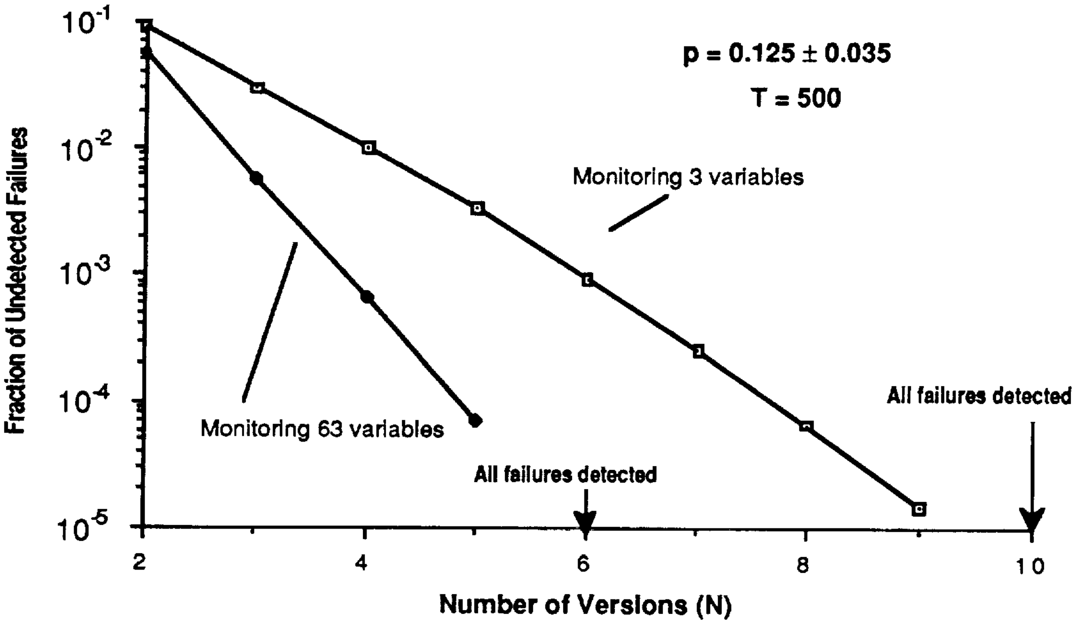


Figure 2 Detecting Failures by Back-to-Back Testing.

Use of either hypertext or a database environment in order to produce a complete semantically linked document (specification body, answers, to questions, relevant reference material) may have possibly avoided errors due to inadequate background knowledge, and unclear specifications (e.g. D1.3, D7, D8). Similarly, use of a specification language that is closely related to the selected programming language may help. For instance, had Ada been used as PDL for redundant components written in Ada, some of the observed failures (e.g. overwriting input variables, D8.4) would have been detected with the supported programming language constructs. Also, use of tools for static data flow analysis of the code would have detected a large group of faults caused by uninitialized variables (e.g. D3.1, D3.4, D3.5).

The selection of a particular software development strategy might have alleviated some of the observed errors. For example, in practice, a "golden" program is usually not available for failure identification, but a possible alternative is back-to-back testing. If back-to-back testing software development methodology were used, it is clear that some of the specification errors and a large number of correlated software failures (including correlated faults), except for those producing IAW answers in every version of a redundant structure, would have been avoided. The majority of faults (over 65 out of about 80 detected) did not consistently induce coincident IAW responses over two or more versions. So, over 80% could have been detected by back-to-back testing of two or three program versions at a time. The detection of the remaining faults would have required more versions. Our experiments show that monitoring of a larger number of output variables helps increase failure detection efficiency of back-to-back testing. In Figure 2 we illustrate this point by showing the fraction of failures that remain undetected over 500 uniform random test cases given that 2, 3, 4, etc. versions which have gone through acceptance testing are selected at random and tested back-to-back [7]

In general, for most of the observed faults suitable test cases can be developed which cause them to manifest as failures in the program outputs, although it is not clear such a test case would have systematically been generated through a well defined test strategy. It is also clear that for any given selected testing strategy, there is always a fault that would have gone undetected.

The need for formalization of every step of the development process cannot be overstressed. Several of the similar faults detected in this experiment could have been avoided if the unit testing had been more formal, and subject to more stringent control, and/or had the appropriate test cases been used by individual development teams. For example, the D8.1 and D8.2 faults, which spanned 11 versions, could be detected either by an inspection of the code, or by a test case based on the existing specification. The problem was that half of the teams simply did not match the explicitly stated functional requirements with their designs and/or code. Another example is the D4 fault for which an inspection and matching between the specifications, the code, and the completeness of the test cases would have detected the problem.

4. Summary

We have briefly described the nature of the faults detected in a multiversion software experiment. Less than 20% of the faults detected so far can be classified as similar. The certification testing greatly reduced the number of similar faults and their span, and it eliminated all specification faults and practically all influence from communication errors. Nevertheless, a number of subtle faults did propagate into the operational testing phase, and their detection required very extensive dynamic and static analysis of the code.

Some fault avoidance and detection strategies that may be helpful in eliminating many of the observed fault types are formalization of the specifications and of unit development and testing, back-to-back testing, and measurement of the adequacy and quality of the system test data.

One of the greatest challenges in a N-version programming is to harness the creativity of people in solving programing problems. We have observed that software developers are not hesitant about parting company with a written specification when they feel that they have a valid generalization to an algorithm or they think that some part of the specification is not logically consistent. The challenge in multiversion programming is to devise a methodology to exploit the resultant diversity in the developed components.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp 9-14, June 1988.

[8] D.E. Eckhardt, A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, and M.A. Vouk, "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," submitted for publication, 1990.

[9] P.R. Lorczak and A.K. Caglayan, "A Large-Scale Second Generation Experiment in Multi-Version Software: Analysis of Software and Specification Faults", Charles River Analytics Inc., Report R8903 under NASA Contract NAS-1-17705, January 1989.

[10] M.A. Vouk, D.F. McAllister, A.M. Paradkar and S.R. Vemulakonda, "Experiments in Fault-Tolerant Software Reliability", Report #5 on NAG-1-667, NCSU, April 1989.