An Empirical Comparison of Software Fault Tolerance and Fault Elimination[1]

Timothy J. Shimeall

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Nancy G. Leveson

Information and Computer Science Department
University of California, Irvine
Irvine CA 92717

Abstract

Reliability is an important concern in the development of software for modern systems. The authors have performed a study that compares two major approaches to the improvement of software — software fault elimination and software fault tolerance — by examination of the fault detection (and tolerance where applicable) of five techniques: run-time assertions, multi-version voting, functional testing augmented by structural testing, code reading by stepwise abstraction, and static data-flow analysis. The study focused on characterizing the sets of faults detected by the techniques and on characterizing the relationships between these sets of faults. Two categories of questions were investigated: (1) comparisons between fault-elimination and and fault-tolerance techniques and (2) comparisons among various testing techniques. The results provide information useful for making decisions about the allocation of project resources, point out strengths and weaknesses of the techniques studied, and suggest directions for future research.

Key words: Assertions, Back-to-back testing, Code reading, Evaluation of software methodologies, Fault elimination, Fault tolerance, N-version programming, Software experiments, Static analysis, Testing

[1]This work has been partially supported by NASA Grant NAG-1-668, NSF Grant DCR-8521398, a MICRO grant co-funded by TRW and the State of California, and the Naval Postgraduate School Research Council

1 Introduction

Reliability is an important concern in the development of software for modern systems. Software reliability improvement techniques may be classified according to the approach they use to deal with faults (source code defects): 1) fault avoidance techniques attempt to prevent the introduction of faults into software during development; 2) fault elimination techniques attempt to locate and remove faults from software prior to its use; 3) fault tolerance techniques attempt to prevent the faults from causing a program to fail.

Because the resources available on any software development project are necessarily limited, choices need to be made about how many and how thoroughly any of the techniques are applied. However, there is little comparative information available. The few empirical studies that have done comparisons have largely compared the techniques solely on the numbers of faults detected by each technique, rather than examining the relationship between the sets of faults detected. This experiment examines the amount of intersection between the sets of faults detected by various techniques in order to reveal limitations of these techniques and suggest research directions to extend their utility. The techniques compared are n-version programming and back-to-back testing, run-time assertions, functional testing augmented by structural testing, code reading by stepwise abstraction, and static data-flow analysis.

The next section surveys related work. Following that, the experimental procedure used in our study is described, some terminology is defined, the results are presented, and the impact of these results in providing direction for future research is summarized in the conclusions.

2 Related Work

2.1 Fault-Elimination Experiments

There are a large number of studies examining software testing. Much of the recent work has focused on assessing the effectiveness of various testing techniques. Myers[25] did a comparative study of functional testing against code reading for fault detection in small FORTRAN programs. The code-reading methodology used was an informal desk check conducted by 3 participants. Myers found a wide variation between individuals, but no significant difference between the performance of the two techniques.

A study by Hetzel[16] compared code reading, structural testing and functional testing in terms of the faults detected by each technique. In that study, 39 experienced subjects tested three PL/I programs ranging in length from 64 to 170 statements. The study found that functional testing discovered the most faults and code reading the least, with structural testing (using a statement-coverage criterion) falling in between. Code reading

detected faults for which test cases are hard to derive, and it detected initialization faults.

Basili and Selby[5] compared code reading by stepwise abstraction with functional testing (using equivalence partitioning and boundary-value analysis) and structural testing (using a statement-coverage criterion) in four small programs (145 to 365 lines long) written in an Algol-like language. Three of the programs contained naturally-occurring faults, while the fourth contained a mixture of naturally-occurring and seeded faults. This study reported that code reading by stepwise abstraction detected more faults than either of the other techniques studied. Statement-coverage testing detected fewer faults than functional testing. The study also compared the types of faults detected by each method using two classification schemes: omission vs. commission and the type of operation in which the fault was present (initialization, control, data, computation, interface or output). Code reading and functional testing detected insignificantly different numbers of each class of faults except interface faults, where code reading detected significantly more, and control faults, where functional testing detected significantly more. In each case, the statement-coverage testing detected either significantly fewer faults of each type or there was no significant difference in the number of faults detected.

Girgis and Woodward[15] compared the fault-detection abilities of four types of testing: weak-mutation testing, data-flow testing, control-flow testing and static data-flow analysis. The comparison used a set of small FORTRAN programs (textbook examples) that were seeded with faults one at a time by an automated tool, then tested until either the seeded fault was detected or the testing criteria were satisfied. The results indicate a large variation in the effectiveness of the testing criteria. Analysis of the experimental data shows an insignificant difference between the four groups, due to the large variation between the criteria in each group. The differences between the individual criteria were significant. The most effective criterion was All-LCSAJs (Linear Code Sequence and Jump). However, this study failed to indicate if this effectiveness result was due to the choice of faults seeded in the programs or to characteristics of the detection techniques (e.g., the seeding strategy may have favored All-LCSAJs). Moreover, the results may have been influenced by the particular testing strategies of each type used.

Ramamoorthy and Ho[27] studied two forms of static data-flow analysis on large FORTRAN programs. Their results confirmed the limitations of static data-flow analysis, but faults were detected during their experiment. In the 2,000 line program analyzed in that study, simple static data-flow analysis (analysis for unreachable code, interface inconsistencies and locally-uninitialized variables) detected four faults. In a separate 23,000 line program, a more comprehensive static data-flow analysis (performing more thorough uninitialized-variable checking, loop-increment checking and analysis for branch anomalies) detected 20 faults.

There have been proposals to use multi-version voting in the testing process [6, 7, 26, 28]. In this method, known as *back-to-back testing*, the vote itself is used as the test oracle, and therefore a larger number of tests can be executed. A study by Bishop et. al.[6] examined

back-to-back testing by varying the specification language, development practices and implementation language used for the versions. Three professionally-developed versions were used and seven faults were detected by back-to-back testing, after initial individual version testing using a series of functional tests. Of the seven faults, two were common between two of the three versions used. No independent method of verification of the results was used so three-way failures could not be detected.

2.2 Fault-Tolerance Experiments

There have been very few experiments that have explored the use of assertions for fault tolerance. A study by Anderson[2] applied recovery blocks, which use assertions to test the system state, to a real-time control system. The code (a professionally-implemented version of a submarine-control program) was 8000 lines long and organized into 14 concurrent activities. The results showed that while assertions were quite difficult to formulate, some reliability improvement was gained through the use of recovery blocks.

Using software from a voting experiment[19], Leveson, Cha, Knight and Shimeall [22] had a set of 24 students insert assertions into eight versions of a Pascal program (varying in length from 400 to 800 lines) in order to detect errors. The results were compared with error detection using 2-version and 3-version voting. The assertions detected errors associated with as many code faults as voting did (but not necessarily the same faults) but were much more reliable in detecting errors — if assertions ever detected the errors associated with a particular fault, they always did whereas, for the most part, voting only detected errors associated with particular faults part of the time.

There have been several experiments investigating the use of n-version programming alone. The first, by Chen[11], provided little information because of difficulties in executing the experiment. The author also wrote 3 of the 7 programs himself. However, it was noted that 10% of the test cases caused failures for the 3-version systems (35 failures in 384 test cases). Chen reported that there were several types of design faults that were not well tolerated in this experiment, in particular missing-case logic faults.

Avizienis and Kelly examined the use of multiple specification languages in developing multi-version software [4]. The reported data indicates that in over 20% of 100 test cases executed, either there was no majority answer or the majority answer was incorrect.

Another experiment, by Knight and Leveson, found that the hypothesis of statistical independence of failures between independently developed programs did not hold [19]. Furthermore, for the 27 programs run on 1,000,000 test cases, an error was not detected by voting three versions in 35% of the cases where an error actually occurred [20].

One of the problems with all of these studies is that most of them have employed uncontrolled experimental designs. Claims have been made about improvements in reliability due to these techniques in comparison with relatively unverified software. It is reasonable to expect that applying some reliability-enhancing technique would produce an improve-

ment over not applying any special techniques. A more realistic comparison is to examine the reliability of multiple versions voted together versus the reliability of a single version with additional reliability-enhancement techniques applied.

Although it was not the original goal, there is a study that provides one data point in this comparison. Brunelle and Eckhardt[10] took a portion of the SIFT operating system, which was written from a formally-verified design at SRI[24], and ran it in a three-way voting scheme with two new (non-formally verified) versions. The results showed that although no faults were found in the original SRI version, there were instances where the two unverified versions outvoted the correct, verified version to produce a wrong answer[2]. Care must be taken in using this data because the qualifications of the implementors of the three versions may be different.

Examining the results obtained by the previous experiments reveals several characteristics of n-version programming. First, the prevalence of coincident failures (observed in every experiment conducted thus far) reduces the effectiveness of multi-version voting in dealing with faults. Second, there appears to be substantial difficulty in getting versions to agree on a consensus result. Even mathematically correct algorithms sometimes produce differing results due to numeric instability[8].

3 Experimental Design

A set of programs written from a single specification for a combat simulation problem are used in the study described in this paper. The specification is derived from an industrial specification obtained from TRW[12]. The simulation is structured as three sets of transformations from the input data to the output data.

The input data consists of 2600-4500 values (depending on the size of the military units modeled) describing the simulated military units, their capabilities, and the environment in which they interact. The first set of transformations converts the input data to an abstract intermediate state. The intermediate state is updated by a second set of transformations in each cycle of simulated time. This second set of transformations is partitioned into six interacting groups, simulating military positioning, movement, observation, attrition, recovery/repair, communication and the terrain/weather effects present within the simulated battlefield. At each cycle of the simulation, each simulated military unit cycles through each of these activities, acting on each other military unit and being acted upon by each military unit. (The reader is encouraged to see the Dobieski paper[12] for more details.)

After a number of cycles (specified in the input data), the output data are produced by the third set of transformations from the final intermediate state. The output data consist of 5-40 values (depending on the size of the military units modeled). Prototype

[2]These results are not reported in the published paper on the experiment, but were obtained through personal communication with one of the authors.

implementations were developed by three individuals in order to evaluate and improve the quality and comprehensibility of the requirements specification before the development of the versions began.

The experiment participants used throughout were upper-division computer science students. One set of participants, students in a senior-level class on advanced software engineering methods, performed all design and implementation activities on the program versions. A disjoint set of participants attempted to detect faults in the programs. All decisions on whether or not to report a section of code as a fault were made by a student participant or by a program written by a student participant. Once the reports were generated (by all techniques), the administrator acted as final arbiter as to which reports identified faults and which were false alarms; this decision was not reported to the students during their participation in the experiment. All participants were trained in the techniques used in the experiment; however, none had applied these specific techniques on any projects prior to this experiment with the exception of previous Pascal programming experience by the implementation participants.

The development activity involved 26 individuals, working in two-person teams. Teams were assigned randomly. The development activity involved preparing architectural and detailed designs for the software, coding the software from those designs, and debugging the software sufficiently to pass the version acceptance test. Of the 13 teams, 8 produced versions that passed the acceptance test within the time requirements of the experiment. The version acceptance test was a set of 15 data sets. The data sets were designed to execute each of the major portions of the code at least once. The acceptance test was not, and was not intended to be, a basis for quality assessment of the code, but rather was a test of whether all major portions of the code were present in some operable form. The goal of the development procedure was to have the versions in a state similar to that of normal software development immediately prior to unit testing.

Table 1 describes the finished versions. The column marked 'Modules' shows the number of Pascal procedures and functions in each version. The size of the source code is given by two figures, source lines and executable lines, with the latter figure omitting blank and comment lines. The mean code length is 1777 lines, with a standard deviation of 435.

The experimental activity involved applying five different fault detection techniques to the program versions: code reading by stepwise abstraction, static data-flow analysis, run-time assertions inserted by the development participants, multi-version voting, and functional testing with follow-on structural testing. The code reading was performed by eight individuals, following the technique described by Linger, Mills and Witt[23]. Each version was read by one person, and each person read only one version. Prior to code reading, all developer comments were stripped from the version source code. This was done to eliminate the inherent inequality of information based on the widely varying amounts of comments in the version source code and to avoid biasing the code readers through the comments in the code.

#	Version Modules	Total Source Lines	Executable Code Lines
1	72	7503	2414
2	56	3452	1540
3	41	1480	1201
4	57	3663	2003
5	28	1834	1544
6	72	3065	2206
7	75	2734	1978
8	57	1896	1331

Table 1: Version Source Profile

The data-flow analysis was performed by implementing and executing an analysis tool based on algorithms by Fosdick and Osterweil[13].

The development participants were trained in writing run-time assertions, given a text-book chapter[3] as a reference and required to include assertions in their versions. The run-time assertions were present during the application of all techniques. If an assertion condition fails, a message is generated.

A "gold" version has been written by the experiment administrator as an aid for fault diagnosis, but this actually just provides another version to check against. In fact, faults in the gold version have been detected. The gold version is not included in the experimental data. It is, of course, possible that failures common to all of the versions, including the gold, will not be detected. This is an unavoidable consequence of this type of experiment.

Functional testing augmented by structural testing was performed on the programs. A series of 97 functional test-data sets were generated from the specification by trained undergraduates. These data sets were planned using the abstract function technique described by Howden[17]. Part of each plan was a description of the program instrumentation needed to view the output of each abstract function. The structural coverage of the functional data-sets was measured using the ASSET structural testing tool[14], and sufficient additional data sets were defined to bring the coverage up to the all-predicate-uses level. This coverage was selected to provide a thorough level of testing within the time constraints of the experiment. The participants used a total of 60 additional data sets to achieve all-predicate-uses coverage in all versions. The number of data sets executed by each individual version varied from 5 to 13, as needed to achieve the required coverage.

Because some of the techniques applied to the programs are open-ended in terms of possible application of resources, it was necessary to attempt to hold relatively constant

the resources allocated to each technique. This was not necessary for those techniques, namely static data-flow analysis and code reading, that have a fixed and relatively low cost. Table 2 contains the amount of human hours and computer hours devoted to each technique. The time devoted to software testing and voting is approximately two calendar months per version for both where we assumed that the computer could execute for 24 hours a day and 7 days a week while humans worked 40 hours per week.

Furthermore, we assumed that many more test cases can be executed when using back-to-back testing with random generation of test cases because of the lack of necessity to apply an independent validation procedure to the outputs although there is also a necessity to write a test-harness program to implement the voting. Writing a test harness is not necessarily a trivial problem when real numbers are involved since different correct results are possible using different (correct) algorithms due to the use of limited-precision arithmetic. Using a tolerance in the comparisons will not solve the problem [8]. In previous experiments, this consistent comparison problem has resulted in time-consuming debugging of correct programs.

	Computer Hours				Human Hours			
Technique	Mean	SD	Min.	Max.	Mean	SD	Min.	Max.
Code Reading	0	0	0	0	36	15	19	60
Static Analysis	40	30	0.5	104	1	0.1	0.75	1.25
Software Test	84	63	36	219	373	4	366	378
Voting	1415	1055	600	3692	6	1	4	8

Table 2: Hours Devoted to Each Technique Per Version

4 Definition of Terms

Before presenting the results, certain key concepts need to be clearly defined. In particular, it is important to understand what a fault is and when it may be detected by each technique. The IEEE standard defines a fault to be "an accidental condition that causes a functional unit to fail to perform its required function."[1] A failure is an output value that varies from the specified standard due to a fault.

If the correction of a section of code eliminates at least one failure, it is counted as a single fault. Several faults could contribute to the failure for a given data set, and several failures could be due to a single fault. For example, a single data set could reveal separate faults dealing with battalion location and observation of one battalion by another. If either of these faults were solely responsible for failures in other data sets, they are counted as

separate faults. If correction of either of these faults eliminates the failure, they are counted as a single fault. This is because faults may sometimes be due to actions distributed throughout the version code. For example, if a failure results from the initialization code not ensuring an assumption made in some calculation code, this is counted as only one fault, although it could be corrected either by changing the initialization code to ensure the assumption or by changing the calculation code to obviate the assumption.

For most of the techniques used in this experiment, determination of when the techniques detect faults is straightforward. Static analysis and code reading identify the fault precisely to a section of code. The other techniques detect errors, which in this experiment were specific enough to allow relatively straightforward tracing to a fault causing that error. For simplicity, we describe these techniques as "detecting faults," when in fact they detected errors that were later traced to faults. A run-time assertion generates reports when faults produce an erroneous run-time state. Testing detects a fault when the test-failure conditions are satisfied due to behavior caused by a fault. The test-failure conditions are explicitly given in the functional test plans and are developed as part of the test data during the structural testing. Faults detected during analysis of the version source code to formulate the structural test data are also considered detected by structural testing.

When considering fault tolerance using multiple-version voting, it is clear that if a correct answer is produced despite the failure of one of the programs, then the triplet is fault tolerant and the single error is masked. If no agreement is reached, then one could say that the individual program failures were detected, but fault tolerance (run-time masking) has not been achieved. In the third case, i.e., producing an incorrect result, the failure is not detected and the faults have not been tolerated. For back-to-back testing, the conditions when a fault is detected are less clear. We define a fault as detected if the version containing it is identified as having failed because its answer differs from a majority of the versions. As a consequence of this definition, two-version voting detects faults whenever the results of the pair disagree. All faults detected by three-version voting are also (by definition) detected by two-version voting, but the converse is not true.

5 Results

Two general categories of questions have guided our analysis of the data. The first is a comparison between fault elimination and fault-tolerance techniques. The second category of questions involves comparing various testing techniques with respect to fault detection, including consideration of their relative strengths and weaknesses and how these techniques might be improved.

5.1 Fault Tolerance and Fault Elimination

To examine whether fault-tolerance techniques are substitutes for fault-elimination techniques, we shall consider the faults tolerated by 3-version voting and potentially tolerated by techniques using assertions. Note that the assertions themselves provide no fault recovery ability, but may be used in conjunction with either forward or backward recovery strategies to tolerate faults once they are detected. No recovery strategies were implemented in the programs used in this experiment. This means that the results relating to assertions should be viewed as counts of faults potentially tolerated (if the hypothetical recovery techniques were effective) rather than faults actually tolerated. This contrasts with fault tolerance by voting, where the same mechanism (a vote) is used to detect and tolerate faults. The results in this section relating to voting are counts of faults actually tolerated by 3-version voting (i.e., where a majority produced a correct result) in this experiment.

The first question that was investigated is whether run-time voting tolerated the faults detected by the fault-elimination techniques used. It has been suggested by Avizienis and Kelly[4] that multiversion voting may reduce or replace traditional software V&V.

> "By combining software versions that have not been subjected to V&V testing to produce highly reliable multiversion software, we may be able to decrease cost while increasing reliability. Most errors in the software versions will be detected by the decision algorithm during on-line production use of the system. The software faults then can be fixed without affecting system availability." [4]

In order to offset the extra cost involved in producing multiple versions, at least one commerical avionics software manufacturer has asked for a reduction in the testing required by the FAA, arguing that the use of multi-version programming will decrease the need for unit testing[21]. In general, using any reliability-enhancing technique has some cost involved with it, and tradeoffs must always be made between using different techniques or combinations of techniques based on cost-benefit analysis.

If voting tolerates the faults detected by testing, then elimination or reduction in testing can possibly be justified, and testing could be completed while the software is being used. However, if the faults detected by fault elimination are *not* tolerated by voting at run-time, then testing cannot be eliminated. Furthermore, any argument for reduction of testing would need to prove that the reduction in testing merely results in the non-detection (and non-elimination) of the faults that voting will reliably tolerate during execution and does not result in run-time failures caused by faults that might have been detected and eliminated by increased testing.

There are two aspects to answering this question. The first is whether the same faults are both detected by the combined fault-elimination techniques and tolerated by voting. A second is whether one particular type of testing is superfluous when using voting because it detects the same faults that voting tolerates.

The programs were executed on 10,000 randomly-generated data sets. In general, we found that the faults that were tolerated were not the same as the faults that were detected by fault-elimination techniques, and the faults that *were* tolerated were not tolerated with high reliability. Table 3 shows the number and intersection of faults found by each class of technique. Twenty-seven faults, given by the sum of the last two lines of table 3, were both tolerated by voting and detected by a fault-elimination technique.

	Version								
	1	2	3	4	5	6	7	8	Total
Tolerated by vote	7	7	11	8	14	7	5	8	67
Detected by assertions only	4	3	1	8	2	2	4	4	28
Detected by fault elim. only	4	17	27	13	15	4	13	26	119
Assert & fault elim.	7	1	1	3	6	3	0	0	21
Assert & vote	0	0	2	0	1	0	2	3	8
Vote & fault elim.	0	2	3	3	2	4	5	2	21
Assert, vote & fault elim.	0	0	1	1	2	1	0	1	6

Table 3: Number of Faults Tolerated or Detected

The tolerance of faults by voting is not as consistent as implied by the data in table 3. In reality, two very different things are being compared. Fault elimination techniques are used to detect faults which are then (hopefully) eliminated before the production use of the software. On the other hand, fault-tolerance techniques may tolerate the errors caused by faults, but the faults remain in the code. Tolerating an error once that is caused by a fault does not necessarily mean that all errors caused by that fault will be tolerated. In the table, run-time voting is credited with tolerating a fault if it tolerates at least one failure caused by that fault even though it may not tolerate every failure caused by the fault. It is also credited with tolerating a fault if only one or several of the 56 combinations of versions tolerate it even though all of them do not. In general, we found that even when the failure caused by a fault is at times tolerated by a triplet, it is usually not tolerated every time, and there is wide variation among the different triplets in terms of how effective they were in tolerating faults.

Of the 56 total voting triplets, the average individual triplet tolerated 33 faults of the 104 faults present in the average individual triplet. The best triplet tolerated 41 faults out of the 107 detected by all techniques in the versions (numbers 4, 5 and 6) that participated in that triplet while the worst tolerated 27 faults out of the 107 detected by all techniques in the participating versions (numbers 1, 5 and 6).

In order to show the variation, we computed the number of faults tolerated at least once by a triplet divided by the total number of faults that caused a failure in one of the

versions comprising that triplet. This fraction ranged from 60.4% to 88.6% with an average of 75.9% and a standard deviation of 6.2%. Note that these are percentages of the faults present in the three versions that constitute the voting triplet and not percentages of all faults found in all versions.

Another way of looking at variability among triplets is to consider the conditional probability that a triplet will mask a failure given that a failure occurs (i.e., the conditional probability that a correct result is produced despite the failure of one of the versions). This fraction ranged from 20.8% to 61.5% with a mean of 37.9% and a standard deviation of 11.1%. On average the triplets only tolerated faults 38% of the time that they caused a failure[3]. This can be explained by the large number of correlated failures that occurred.

The other side of the above question is whether there were any faults tolerated by run-time voting that were not detected by the fault-elimination techniques. If so, then the use of fault elimination does not preclude the use of fault tolerance, i.e., they are complementary techniques rather than competitive techniques. Again, table 3 shows that this did occur for 67 faults, although again it must be remembered that the errors caused by these faults were not tolerated very reliably. Firm conclusions cannot be drawn from this data given the novice nature of the participants in the fault-elimination efforts, but it does raise interesting questions for further study.

An important related question involves the relationship between coincident failures and testing. There has been speculation about whether the faults that result in coincident failures (and thus reduce the fault-tolerance capability of voting systems) are likely to be detected through testing procedures. Examination of the specific faults that were detected by testing indicates that testing detected only 24 of the 103 coincident-failure faults. Furthermore, the coincident-failure faults found by testing did not include those faults that produced the majority of the coincident failures during execution.

5.2 Comparison of Fault-Detection Techniques

Some comparison of the fault-detection techniques is possible with this data, although absolute numbers may not be important because of the problems of evaluating and keeping constant the amount of effort put into each technique. Furthermore, numbers are not really the issue; instead a more important question may be whether different or similar faults are found by each technique. A technique may only find one fault, but if that fault is not likely to be found in any other way, then that technique may still need to be applied.

The reader should note that we are now reinterpreting our experimental procedures. In the previous section, we identified the execution of the 10,000 input cases as a simulation of the production use of the software. We are now interpreting this procedure as a fault-elimination technique that would precede the actual production use of the programs. There

[3]Another way of saying this is that, on average, the triplets only tolerated 38% of the failures that occurred.

is no problem with this from an experimental design standpoint since the procedures are identical and differ only in the time they are performed.

5.2.1 Fault Detection Summary

Table 4 shows the number of faults detected by each technique. Note that the figures in table 3 and table 4 are not comparable due to the difference between fault detection by voting and fault tolerance by voting and the difference between 2-version and 3-version voting. The first five lines give the number of faults detected by each technique that were detected by none of the other techniques (e.g., run-time assertions detected a total of 23 faults that were not detected by voting, testing, static analysis or code reading). The remainder of the table gives the number of faults detected in common by the techniques named on each line (e.g., code reading found a total of 4 faults that were also found by run-time assertions, but were not found by any other technique).

The voting technique used in constructing table 4 was two-version voting. Three-version voting detected 112 of the 123 faults found by two-version voting. These faults were found by voting with the eight versions combined into the 28 possible pairs and the 56 possible triplets. The values in the line marked '2-Version voting (high/low)' in table 4 are the maximum and minimum of the faults detected by each two-version voting pair for each version. A range exists for voting because it was applied to each version 7 times (the number of two-version voting systems in which each version participated) while the other techniques were applied only once. In all other voting cases (voting in combination with each of the other techniques), there were no variations in the number of faults detected. The interesting feature of table 4 is not the precise values shown (which depend on the application), but that most of the faults detected by each technique were found by no other technique.

5.2.2 Variation in Voting Performance

To give some feeling about the variability of the voting performance, two sets of statistics are provided. The first set is the number of faults detected at least once by each pair and each triplet, divided by the total number of faults that caused at least one failure in the versions making up the system (i.e., the fraction of revealed faults that each voting system detected). For the 28 two-version voting systems, the fraction of faults detected varies from 91.3% to 100% with a mean of 97.9% and a standard deviation of 2.6%. For the 56 three-version voting systems, the fraction of faults detected varies among the triplets from 90.5% to 100% with a mean of 96.5% and a standard deviation of 2.5%. In short, the majority of pairs and triplets fail to detect at least some of the faults revealed by the input data. Since virtually all systems would be developed with at most one pair or triplet (not the 28 and 56, respectively, that we had), the data in table 4 represents a best case

	Version								
	1	2	3	4	5	6	7	8	Total
Faults detected by precisely one technique:									
Testing	2	12	21	11	13	1	11	10	81
2-Version voting (high/low)	$\frac{10}{11}$	$\frac{9}{9}$	$\frac{11}{12}$	$\frac{7}{8}$	$\frac{14}{14}$	$\frac{7}{8}$	$\frac{5}{6}$	$\frac{10}{10}$	$\frac{73}{78}$
Code reading	0	2	4	2	0	1	0	16	25
Assertions	3	3	1	8	1	1	3	3	23
Static analysis	0	0	2	0	0	0	0	0	2
Faults detected by precisely two techniques:									
2-v. voting & test	3	1	1	3	2	6	4	0	20
Assertions & test	5	1	0	2	5	3	0	0	16
Assertions & 2-v. voting	0	0	2	0	2	0	4	4	12
Reading & assertions	2	0	0	1	1	0	0	0	4
Static analysis & 2-v. voting	0	0	0	0	0	1	1	0	2
Reading & 2-v. voting	0	0	2	0	0	0	0	0	2
Reading & test	0	0	0	0	0	0	1	0	1
Static analysis & test	0	0	0	0	0	0	1	0	1
Faults detected by precisely three techniques:									
Assert & 2-v. voting & test	0	0	1	1	2	1	0	0	5
Reading & 2-v. voting & test	0	2	0	0	0	0	0	2	4

Table 4: Number of Faults Detected

for voting.

The second set of statistics used to analyze the variation in fault detection is the conditional probability that a pair or triplet detects each fault given that it is revealed. The mean of these probabilities over all faults detected for each two-version system varies between 0.826 and 0.981, with a mean of 0.934 and a standard deviation of 0.040. In other words, voting with a two-version system never detected all of the failures of the component versions. For the three-version voting systems, the mean of the conditional probabilities over all faults detected by each system varied from 0.787 to 0.953 with a mean of 0.886.

5.2.3 Back-To-Back Testing

There are two particularly interesting comparisons to make that deal with currently unresolved issues in testing research. The first is the use of back-to-back testing vs. the use of other testing oracles (i.e., those not involving a voting procedure). Back-to-back testing allows a large amount of data to be executed due to the automated nature of the oracle, and it has been advocated as a way of extensively testing complex software where determining a correct answer by a non-voting procedure may be tedious and time-consuming[6, 7, 26, 28]. Of course, if one takes a large perspective, part or all of the savings in testing may be offset by the cost of producing and debugging multiple versions of the software. However, if back-to-back testing is much more effective then the cost arguments may be irrelevant.

The underlying assumptions in back-to-back testing are that (1) given that a fault leads to an erroneous output it will be detected by the voting process, and (2) the faults that would have been detected by other testing techniques, such as structural testing or static analysis techniques, will be elicited and detected by voting on random or functional test cases alone. Both of these assumptions can be checked with our data.

There were 78 faults that were detected by the voting procedure that were not detected by any other technique. Even given the novice nature of the participants in the other testing procedures, they found 153 faults that were not detected by the back-to-back testing. Forty-five faults were detected in common. There were faults that did not cause failures on the randomly-generated test data and therefore could not possibly have been detected by the back-to-back testing, but were found by the techniques that do not require failure to detect faults.

A related question is whether better results are obtained by doing the back-to-back testing on both randomly-generated test cases and functionally-generated test cases. This separates the issue of test data generation from the issue of using voting as a test oracle. We executed the 56 triplets on the functionally-generated and structurally-generated test cases and did not detect any additional faults. This implies that the problem is not necessarily in the test case generation method, but in the identification of errors by voting, i.e., by the limitations of using voting as a test oracle.

5.2.4 Types of Faults Detected

To examine the fault detection behavior of the techniques further, the types of faults detected by each were profiled and compared. Because there is no widely-accepted, detailed taxonomy for fault classification, a 13-class fault taxonomy was developed and used. This taxonomy was developed by studying the faults detected and then abstracting the common features of the faults detected by specific techniques. As a result, the taxonomy must be considered to be relatively arbitrary.

The fault taxonomy is described in table 5. Since the taxonomy was developed to differentiate between the sets of actual faults detected by the various techniques, significance tests on these classifications are inappropriate. In the following discussion, a statement that a technique detected faults in a particular class does not imply that the technique detected all faults of that class. For example, the statement that voting detected missing-thread faults should not be interpreted as indicating that all missing-thread faults located in the versions were detected by voting. Two of the categories in this fault taxonomy (Overrestriction faults and Data-Structure faults) are not mentioned in the text that follows since these categories do not provide a basis for characterizing the differences observed in fault detection by the various techniques.

Code Reading by Stepwise Abstraction. Code reading by stepwise abstraction found calculation faults, missing-check faults, branch-condition faults and missing-branch faults. The participants did not find large global pieces of missing code or missing threads of logic that ran through the entire program.

Analysis of the experiment data lent insight into two questions related to the use of code reading in software development. The first of these questions is what conditions led code reading to fail to detect faults. One reason for the failure of code reading to detect certain faults was the omission of needed detail in the abstractions constructed by the participants. A condition that seemed difficult to detect was missing code. The code-reading participants detected missing-branch faults (i.e., where the set of cases handled by the code did not cover all possible cases at a specific point), but failed to detect those cases where larger or more widespread code was omitted.

A second question of interest in examining the code-reading results is what conditions led the code-reading participants to erroneously report code as faulty. Analysis of the annotations written by the code readers indicates that false alarms arose from code that was difficult to abstract. For example, an erroneous fault report was generated for a procedure with a large number of arguments that was called in several places in the code. Some of the formal parameters were used in different ways in the procedure (depending on the value of other formal parameters), and this led to misconceptions on the part of the reader. False alarms also occurred when abandoned implementation strategies (blind alleys during development) are reflected in the code. For example, the readers erroneously

Class	Comments	Detecting Technique
Overrestriction	E.g., forcing all weather to move north-east, rejecting legal input	Assert, Read, Test, Vote
Loop Condition	E.g., infinite loop	Vote, Assert, Test
Calculation	Incorrect formula	Read
Initialization	Variable not initialized	Stat. Analysis, Test
Substitution	Wrong variable used	Vote, Assert
Missing check	Exceptional case not handled E.g., divide by zero	Read
Branch Condition	Bad condition on a branch	Vote, Read, Test
Missing Branch	Localized missing code to detect and handle specific conditions in normal execution	Read, Test
Missing Thread	Missing path throughout program	Vote, Test
Unimplemented Requirement	Missing functionality on all paths	Test
Ordering	Operations in wrong order (e.g., updating value before use)	Vote, Test
Parameter Reversal	Actual parameter order permuted with respect to formal parameter	Vote, Assert
Data Structure	E.g., linked list becomes circular	Vote, Test, Read, Assert

Table 5: Fault Taxonomy

reported several faults in cases where the name of a variable conflicts with the manner in which the variable is used. Annotations by the code readers in such situations indicate that they focused on syntactic factors rather than the program semantics. These results suggest that experience and improved training may help reduce erroneous reports from code reading.

Use of commented code for the code reading might have prevented some of the false alarms of the readers. However, it might also have led to different types of misdirection. For example, it is equally possible that the readers would report code as erroneous when it conflicted with the comments (e.g., where the development participants corrected the code, but not the comments describing the code). Another possibility is that the readers would have failed to detect as many faults, e.g., if they summarized the comments instead of the code and thus duplicated the faulty assumptions made by the development participants.

Static Data-Flow Analysis. Static data-flow analysis found only initialization faults. Three faults were found by this technique that were not detected by any other. Upon examination, it was determined that the compiler and operating system versions being used happened to initialize to zero the particular storage locations where the programs were loaded, and these variables were used for counters and needed to be initialized to zero. Obviously, this cannot be counted on in future versions of these support programs so these are real and important faults to detect.

Voting. Voting found missing-thread faults, parameter-reversal faults, substitution faults, ordering faults and faults (subsets of loop-condition and data-structure faults) causing abends (which, despite their cause, are obviously found by any of the techniques that involve executing the code over a large number of test cases).

It is interesting to consider the faults that were not found by voting, i.e., those that were so highly correlated that the faults were masked by the voting procedure. For the most part, these were missing-branch faults. This is consistent with past experiments, which have all reported that missing-logic errors are poorly tolerated by multi-version systems. Testing strategies, such as functional and structural testing, that examine special cases as well as typical cases were more successful at finding missing-branch faults. As discussed above, performing back-to-back testing on the test cases derived for functional testing did not solve the problem since the common faults masked the identification of the fault even though the programs failed.

Another unmasked fault involved the use of a wrong subscript. This is puzzling as the same thing happened in a previous experiment [9]. We cannot currently find any other explanation aside from coincidence.

Run-Time Assertions. Run-time assertions found parameter-reversal faults, substitution faults and faults causing abends. They did not detect any of the four classes of missing-code faults. We are not very confident about the data for run-time assertions as the programmers involved did not have any experience in writing exception or error-detection code, and our subjective evaluation of their assertions is that they were, in general, quite poor. All of the run-time assertions used were simple range or specific value tests (e.g., run-time assertions to check if the variable `Params.NumWeatherEvents` lies between 0 and the constant `MaxWeather`, or if pointers are non-nil); consistency of internal results were not checked. In fact, examination of the design documents show cases where the development participants anticipated faults that actually occurred in their code, but (for reasons known only to them) they omitted assertions to check for these faults.

Despite these weaknesses, the simple range checks detected 23 faults that were found in no other manner. The fact that assertions detected faults that voting did not is consistent with the results of our previous study of assertion effectiveness[22]. It appears that even a cursory set of assertions has some value, and this suggests that it would be useful to perform further work to examine the effectiveness of a thorough set of assertions for fault detection.

Functional and Structural Testing. Functional and structural testing identified ordering faults, missing-branch faults, unimplemented-requirement faults and missing-thread faults. They also detected faults causing abends (as did all the techniques that involved executing the programs). Structural testing detected further missing-code faults, in particular faults involving variables that were initialized in a manner that in rare cases conflicted with the manner in which those variables were used.

Structural testing failed to detect several missing-thread faults that were found by other techniques (such as voting). The incompleteness seemed largely due to the module-by-module nature of the testing tool used. That is, the prototype version of ASSET used in this experiment measures the coverage achieved by the input data on each module individually, with no consideration of data flow between modules. The versions contain several instances where global data structures are initialized in one module, updated in a second module and used in calculations in several other modules. While all of the initialization paths and all of the update paths are covered by the test data, not all of the update paths are covered for each initialization path. Therefore, several of the missing-code faults eluded detection in our structural testing.

Examination of the functionally-specified test data sets showed that faults were revealed only by those data sets that contained atypical data (i.e., those tests that exercised special cases in the versions or odd combinations of the functions supported by the code). This result supports a recommended practice in the field of software testing.

5.2.5 Additional Comparisons of Fault Detection Techniques

Two general attributes accounted for much of the observed variation of effectiveness: the ability of the techniques to examine internal states and the scope of their evaluation.

One reason voting failed to detect some faults was that it was not able to examine internal program states. The other techniques do not share this limitation. For code reading and static analysis, examination of the internal state involves evaluation of the program source code. Functional testing identifies and evaluates internal abstract functions. Assertions evaluate specific internal conditions at the locations where they are inserted. Because the voting systems examine only final states, they fail to identify faults that occur, but are concealed by later processing.

Tso and others argue that voting may be performed on internal program states, as in the cross-check analysis technique[29]. However, the programs in this experiment are quite diverse. The internal program states differ significantly in the algorithms and data structures employed. A single value in the internal state of one program may indeed be a single value in another program, but more often it is either a function of several values or not present at all (unneeded in the alternate algorithm used in the second program). Furthermore, since the order of the programs' operations are also quite diverse, there is no single time except initialization and production of the final result at which any correspondence in values could be compared by voting. To allow voting on internal program states requires specification of the algorithm and data structures used in the internal states, effectively eliminating any significant design diversity and thus eliminating the ability to detect design errors.

A second important attribute is the scope of evaluation. The scope over which assertions and code reading examine the system state appears to be the key characteristic limiting the detection of faults by those techniques. Assertions examine the system state at specific points in the execution. If a fault has not yet occurred at those points, or if the fault's effect is masked, the assertion does not detect the fault. The use of code reading by stepwise abstraction on the uncommented form of these programs did not detect certain faults because the process of abstraction did not maintain sufficient detail. For example, an assumption by the coder might be violated by the faulty code, but the assumption is not preserved through several layers of abstractions made by the reader between the initialization and the calculation code. The purpose of abstraction is to keep the amount of information at a manageable level, but over-abstraction limits the effectiveness of code reading.

6 Conclusions

It is important to consider several caveats when drawing conclusions from the data presented in this paper. First, experts in the various techniques were not used. Students get

a lot of experience in programming while in school, but they seldom receive adequate exposure to and practice with testing and other fault-elimination techniques. We gave them training, but that is not a substitute for experience. Furthermore, only one method was applied within each category of fault-elimination techniques; the particular method chosen may not have been the most effective. Finally, our program may not be representative of a large number of applications and the particular software development procedures also may not be representative.

Despite these limitations (which unfortunately are inherent in this type of experimentation), useful information can be derived from this study. In the few instances where there is other experimental evidence, our results tend to support and confirm previous findings. Where almost no experimental evidence is available, our results represent one data point that can be used to focus and direct future experiments.

This experiment is the first to investigate the relationship between fault-elimination techniques and software fault tolerance. We found that our data does *not* support the hypotheses that multi-version voting is a substitute for functional testing, that testing can be reduced when using this software fault-tolerance technique, nor that testing can proceed in conjunction with operational use of the software in an n-version programming system where high reliability is required. Instead, we found that multi-version voting did not tolerate most of the faults detected by the fault-elimination techniques and was unreliable in tolerating the faults it was capable of tolerating. Although we also found that multi-version voting tolerated different faults than were detected by the fault-elimination techniques, no firm conclusions should be drawn from this because of doubts about the ability of the novices involved and the limitations of the fault elimination techniques used; further investigation is suggested.

The testing in this experiment largely failed to detect the faults responsible for coincident failures of multiple versions. This result occurred despite the fact that some of the testing techniques target the testing of special cases, which are often involved in coincident failures. This result may indicate that the faults that reduce the effectiveness of n-version programming are among the most difficult to detect. This is not surprising and satisfies the intuitive explanation that the parts of the problem that lead to mistakes by the programmers may be equally difficult for the testers to handle. The whole field of psychology is predicated on the assumption that human behavior (including that involved in making mistakes) is not random.

The experiment also examined a broad set of fault-detection techniques in a comparative manner. While the presence of multiple versions can speed the execution of large numbers of randomly generated cases, our results cast doubt on the effectiveness of using voting as a test oracle. Testing procedures that allow instrumenting the code to examine internal states were much more effective. When comparing fault-elimination methods, we found that the intersection of the sets of faults found by each method was relatively small. Examination of the faults allowed us to categorize the types found by each method and, in some cases,

to explain why these results occurred.

This experiment raises questions with respect to several of the techniques examined. The detection capability of code reading appears to be reduced in comparison to earlier results such as those reported by Basili and Selby[5] (who used smaller programs). Additional research is needed to distinguish the effects of program size and complexity on the effectiveness of code reading. Analysis of the faults not detected shows that there is a need to develop extensions to code reading techniques that better characterize global effects. One way of accomplishing this might be to mix a top-down code reading technique with the bottom-up methodology of code reading by stepwise abstraction. Further investigation of this seems worthwhile.

The static data-flow analysis technique used in this study is limited in the type of faults it can potentially detect. However, several of the faults found by this technique were found by no other technique, and so applying it in software development may be worthwhile, particularly given its relatively low cost of application. There may also be language or environmental factors that reduced the number of undefined reference faults in this particular software. For example, the requirement for declaring all variables in Pascal may serve as a reminder to initialize variables before use. Other static-analysis techniques, such as associating physical units with variable values and analyzing the software to see if the units are appropriately preserved[18] deserve further exploration. These types of techniques would permit examination of the legality of usage rather than just the presence of initialization and reference.

7 Acknowledgement

George Dinsmore and Robin Kane of TRW Corporation and Stephanie (nee Leif) Aha aided in many different aspects throughout the work described. Phyllis Frankl and Elaine Weyuker provided the ASSET testing tool. Debra Richardson and Richard Selby provided many constructive comments on earlier reports of this study. The reviewers also provided many useful comments that improved this paper.

References

[1] "Glossary of Software Engineering Terminology", ANSI-IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[2] T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding, "Software Fault Tolerance: An Evaluation", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1502–1510.

[3] T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice*, Prentice Hall, New Jersey, 1981.

[4] A. Avizienis and J.P.J. Kelly "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, August 1984, pp. 67–80.

[5] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, December 1987, pp. 1278–1296.

[6] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahl and J. Lahti, "PODS – A Project on Diverse Software", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, (1986), pp. 929-940.

[7] S.S. Brilliant, *Testing Software Using Multiple Versions*, Ph.D. Dissertation, University of Virginia, Charlottesville, VA, September 1987.

[8] S.S. Brilliant, J.C. Knight and N.G. Leveson, "The Consistent Comparison Problem in Multi-Version Software", *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, November 1989, pp. 1481-1485.

[9] S.S. Brilliant, J.C. Knight and N.G. Leveson, "Analysis of Faults in an N-version Software Experiment", *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 2, February 1990, pp. 238-247.

[10] J.E. Brunelle and D.E. Eckhardt "Fault Tolerant Software: Experiment with the SIFT Operating System," *AIAA Computers in Aerospace V Conference*, October 1985, pp. 355–360.

[11] L. Chen and A. Avizienis, "N-Version Programming: A Fault Tolerance Approach to the Reliability of Software", *Eighth Int. Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1978, pp. 3–9.

[12] A. W. Dobieski, "Modeling Tactical Military Operations", *Quest*, Spring 1979, pp. 1–25.

[13] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability", *ACM Computing Surveys*, Vol. 8, No. 3, September 1976, pp. 305–330.

[14] P. Frankl and E. Weyuker, "Data Flow Testing in the Presence of Unexecutable Paths", *Workshop on Software Testing*, Banff, Canada, July 1986, pp. 4–13.

[15] M.R. Girgis and M. R. Woodward, "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria", *Proceedings of the Software Testing Workshop*, Banff, Canada, 1986, pp. 64–73.

[16] W.C. Hetzel, *An Experimental Analysis of Program Verification Methods*, Ph.D. Dissertation, University of North Carolina at Chapel Hill, 1976.

[17] W.E. Howden, "Functional Testing and Design Abstractions", *Journal of Systems and Software*, Vol 1, 1980, pp. 307–313.

[18] W.E. Howden, "A Survey of Static Analysis Methods", *Tutorial: Software Testing and Validation Techniques*, IEEE Press, 1981, pp. 101–115.

[19] J.C. Knight and N.G. Leveson, "Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Transactions on Software Engineering*, January 1986, pp. 96–109.

[20] J.C. Knight and N.G. Leveson, "An Empirical Study of Failure Probabilities in Multi-Version Software," *Sixteenth Int. Symposium on Fault-Tolerant Computing,* Vienna, Austria, July 1986, pp. 165–170.

[21] Personal communication with Larry Yount of Honeywell/Sperry Avionics and Robert Hall and Michael Dewalt of the FAA.

[22] N.G. Leveson, S.S. Cha, J.C. Knight and T.J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study", *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 4, April 1990.

[23] R.C. Linger, H.D. Mills and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979, pp. 147–212.

[24] P.M. Melliar-Smith and R.L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight-Control System," *IEEE Trans. on Computers*, Vol. C-31, no. 7, July 1982, pp. 616–630.

[25] G.J. Myers, "A Controlled Experiment in Program Testing and Code Walk-Throughs/Inspections," *Communications of the ACM*, Sept. 1978, pp. 760–768.

[26] C.V. Ramamoorthy, Y.K., Mok, E.B. Bastani, G.H. Chin and K. Suzuki, "Application of a Methodology for the Development and Validation of Reliable Process Control Software," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 6, November 1981, pp. 537–555.

[27] C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 46–58.

[28] F. Saglietti and W. Ehrenberger, "Software Diversity — Some Considerations about its Benefits and its Limitations," *Safecomp '86*, Sarlat, France, October 1986.

[29] K.S. Tso, A. Avizienis and J.P.J. Kelly, "Error Recovery in Multi-Version Software Development", *Safecomp '86*, Sarlat, France, October 1986, pp. 43–50.