An Empirical Comparison of Software Fault Tolerance and Fault Elimination

Timothy J. Shimeall
Nancy G. Leveson
Department of Information and Computer Science
University of California, Irvine

1 Introduction

Reliability is a pressing concern in the development of software for modern systems. Many techniques have been proposed to improve software reliability. One technique, N-Version Programming[9], has been used in software to control aircraft[23] and railroads[14] and has been proposed for nuclear power plants[13]. In this technique, several versions of a piece of software are implemented by different teams of programmers. The versions are all executed in parallel, and their results are collected and voted to provide the system result. One drawback to the n-version technique is that the total development costs are increased due to the costs of developing multiple versions. Since in most situations there is a fixed amount of resources that can be invested in building the software, there must be saving somewhere else in order to allow multiple version development.

In order to make the technique affordable, it has been suggested that n-version programming will be so effective that it can be used as a partial substitute for current software verification and validation procedures[1]. Instead of extensive V&V of a single program, Avizienis and Kelly[2] suggest that independent versions can be executed in an operational environment and V&V completed while using the software. Furthermore, by employing hobbyist programmers to write multiple software versions for critical systems at home, they suggest it will be possible to relax the need for rigorous quality control and centralized programming environments. Although it is difficult to believe that anyone would take these latter suggestions seriously, the need to reduce costs when developing multiple versions has led to at least one attempt to reduce testing of safety-critical software for commercial aircraft based on the argument that the use of n-version programming provides such high reliability that unit testing can be reduced. It seems important to investigate the hypothesis that testing can be reduced in n-version systems and, in general, to study the relationship between fault elimination techniques and fault tolerance techniques.

There have also been proposals to use n-version voting in the testing process [4,5,22,24]. In this method, the vote itself is used as the test oracle, and, therefore, a larger number of tests can be executed. The underlying assumptions here 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.

The authors of this paper are engaged in a large-scale experiment comparing software fault tolerance and software fault elimination as approaches to improving software reliability. This paper describes the experiment and the results that apply to the appropriateness and underlying assumptions of these two proposals, i.e., reducing standard testing procedures when using voting to achieve fault-tolerance in operational software and using voting in the testing process.

2 Related Work

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. Hetzel[15] compared code reading, structural testing, and functional testing with respect to the faults detected by each technique. Code reading detected the fewest faults, but it was found to be effective for initialization faults and faults for which test cases were hard to formulate. Functional testing found the most faults in that study. Myers[21] compared testing against code reading. He found a wide variation between individuals but no significant difference between the performance of the two techniques. Basili and Selby[3] compared code reading against functional and structural testing, finding that code reading detected more faults than the other techniques and functional testing performed better than structural testing.

All of these studies used small pieces of software (less than 400 source lines) to make their comparisons. Given the contradictory results, it appears that no simple rules exist for choosing among these testing techniques. Furthermore, while relative comparisons of number of errors detected provide some basis for choosing between mutually exclusive alternatives, this is not necessarily the situation with respect to testing. Although limited resources and time usually forces limitations in the total amount of testing performed, one would probably want to apply more than one approach for detecting software faults. It would be helpful to have information on the degree to which two techniques are complementary, i.e., detect different errors, or redundant, i.e.,
detect the same errors, along with more detailed information about the particular errors (and hopefully classes of errors) detected and not detected. Some of this information can be derived by theoretical analysis while some will require empirical study since human behavior and capabilities are involved for which few adequate models exist.

There have been several experiments involving the use of n-version programming. The first, by Chen[9], provided little information because of difficulties in executing the experiment. 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[2] examined the use of multiple specification languages in developing n-version software. The reported data indicates that in over 20% of 100 test cases executed, the three version-systems were unable to agree or voted on a wrong answer. In addition, 11 of the 18 programs aborted on invalid input. Despite these results, they conclude that "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." The data in this experiment does not support the hypothesis implicit in this statement that high reliability will be achieved by using this technique. They continue to say that "Most errors in the software versions will be detected by the decision algorithm during on-line production use of the system." There were 816 combinations of the programs in this experiment, each run on 100 test cases for a total of 81,600 calculated results. In 5.6% of the cases where an error occurred in at least one version, the error was not detected by the voting procedure.

Another experiment, by Knight and Leveson[18], found that with 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. The individual programs in this experiment had a much higher average reliability than in the Kelly experiment (i.e., 0.99993 versus 0.72) indicating that they were more thoroughly tested before being subjected to the voting procedure. There is some justification for hypothesizing that faults leading to correlated failures are more likely to escape detection during testing than faults that do not lead to correlated failures, but there has been no previous experiments that explored the relationship between testing and n-version programming.

Knight and Leveson[17] investigated the problems of common failures between independently produced versions and have also looked at reliability improvements[18]. Although the failure probability was decreased (about 10 times) using three-version voting compared to single version in the latter study, this comparison is not a realistic one. It is reasonable to expect that applying some reliability-enhancing technique would produce an improvement 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 resources applied to enhance the reliability.

Although it was not the original goal, there is a study that provides one data point in this comparison. Brunelle and Eckhardt[8] took a portion of the SIFT operating system, which was written and formally verified at SRI[20], and ran it in a three-way voting scheme with two new (non-formally verified) versions. The two new versions were actually more efficient than the original version. The results showed that although no faults were found in the original formally-verified SRI version, there were instances where the two unverified versions outvoted the correct, verified version to produce a wrong answer.

Care must be taken in using this data because the qualifications of the implementors of the three versions may be different. Nevertheless, the results are interesting. The authors are not aware of any studies that have produced data comparing n-version programming and testing.

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[10]. The simulation is structured as three sets of transformations from the input data to the output data. The first set of transformations converts the input data to an abstract intermediate state, which is updated by a second set of transformations in each cycle of simulated time. 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. Prototype 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 experimental subjects used throughout were upper-division computer science students. They were selected on the basis of interviews and a review of their transcripts (except for the programmers who were students in a senior-level class on advanced software engineering methods). 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. Participants submitted questions on the specification or on the application of the techniques through electronic mail and received standardized individual replies via the same medium. On two occasions during development the specification was revised to correct vague wording and typographical errors. Revision of the specification was performed by published errata sheets. All participants also submitted background profiles and prepared time-sheets documenting their efforts.

The development activity involved 26 individuals, working in two-person teams. Teams were assigned randomly from students in an upper division computer science course. All development participants were senior-level computer sci-

---

Footnote:

1 These results are not reported in the published paper on the experiment but were obtained through personal communication with one of the authors.
ence majors. Of the 13 teams, 8 eventually produced versions that were judged acceptable for use in the experiment. The development activity involved preparation of architectural and detailed designs for the software, coding the software from those designs, and debugging the software sufficiently to pass the version acceptance test. 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. The final programs vary in length from 1186 to 2489 lines of Pascal code.

The experimental activity involved applying five different fault detection techniques to the program versions: code reading by stepwise abstraction[19], static data reference 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. Each version was read by one person, and each person read only one version. The data reference analysis was performed by implementing and executing an analysis tool based on algorithms by Fosdick and Osterweil[11].

In addition to code-reading and static analysis, the source code for each version has been executed using 10,000 randomly-generated test cases. The test data generator has been designed to provide realistic test cases according to an expected usage profile in the operational environment. The development participants were trained in writing run-time assertions and were 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. Failures that do not result in abnormal termination of the programs may be detected also by comparison of the eight version outputs.

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[16]. Part of each plan was a description of 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[12], and sufficient additional data sets were defined to bring the coverage up to the all-predicate-uses level.

An extremely large amount of data has been collected, and analysis of this data is currently underway. This paper reports on some results involving comparison of the fault elimination techniques (except for structural testing) and fault tolerance techniques applied to the programs.

4 Results

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 reference analysis and code reading, that have a fixed cost. Table 1 contains the amount of human hours and computer hours devoted to each technique. The time devoted to functional testing and back-to-back testing is approximately two calendar months per version for both.

<table>
<thead>
<tr>
<th>Method</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Analysis</td>
<td>40</td>
</tr>
<tr>
<td>Code Reading</td>
<td>0</td>
</tr>
<tr>
<td>Functional Test</td>
<td>4</td>
</tr>
<tr>
<td>Back-to-Back Testing</td>
<td>1415</td>
</tr>
</tbody>
</table>

Table 1: Hours Devoted to Each Method Per Version

In general, the student participants took their efforts seriously and performed careful work in this experiment. In the code reading, the rate of analysis and annotation was somewhat surprising. Given the length and complexity of the software, the average rate of 70 lines per hour was higher than expected. However, analysis of the annotations shows that the students were reading in depth and not simply skimming the source listings. In the functional test planning, similar attention to detail was displayed by the students. It is possible for test plans to be of variable quality, so three outside industrial experts evaluated the plans in an attempt to gauge the plan quality before testing the versions. In their opinion the plans, while obviously not the work of experienced professionals, were comparable to those used in current industrial settings.

Two general categories of questions have guided our preliminary analysis of the data. The first is a comparison between fault elimination and fault tolerance techniques, i.e., are they substitutes for each other or do they complement each other. The second category of questions involves comparing various testing techniques with respect to fault detection including the use of n-version voting as a test oracle.

4.1 Fault Tolerance vs. Fault Elimination

Before presenting the data, it is necessary to define some terms. The term "run-time voting" will signify the process of voting three versions at a time on a set of input cases representing an operational profile. We chose three versions
because more than this number of versions would, in most cases, be totally unrealistic, and two-version voting does not provide fault tolerance. When at least one version in a two-version system provides an erroneous result, there is no chance of masking (or tolerating) that failure and either no answer or a wrong answer is provided.

The first question investigated was whether voting during production use of the software provides high enough reliability to justify reducing normal software verification and validation. Not only must the voting detect errors, but there must be a reasonable level of reliability (defined as correct answers produced) to justify using the software in a relatively unverified state, especially if erroneous outputs can have costly consequences, as described above. This was not the case.

In a three-version voting system, there are three possible results: a correct answer is produced (there are at least 2 programs in the triplet that produce correct answers), no answer is produced (three different results are produced), or a wrong answer is agreed upon (at least two programs produce identical wrong answers). 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 the error (or errors) were not masked. The third case, i.e., producing an incorrect result out of the triplet, is the most potentially dangerous or costly.

Table 2 compares the results for the individual versions and the 56 triplets. The percentage of cases executed correctly by the individual versions varied from 40.0% to 92%, with a mean of 74.4% and a standard deviation of 18.6%. While this may seem low, it should be noted that these programs were non-trivial and that the acceptance test was not designed to enforce a high reliability level since that was deemed atypical of normal software development prior to unit testing. The software developers had, supposedly, debugged their versions to the best of their ability before submitting them. Note that in the table, the probability of no answer for a single version is equivalent to the program normally terminating due to a run-time failure. There was no attempt to include exception-handling in the individual versions, so a comparison between run-time error detection by single versions and error detection by the triplets is not meaningful.

Production use of the software was simulated by voting all possible triplets on 10,000 inputs generated according to a random but realistic operational profile. Three-way voting of these versions produced correct results an average of 81.1% of the time with a standard deviation of 9.4%. This is not significantly different from the mean for the individual versions, and it is certainly far from being of acceptable reliability for most applications. That is, there was no statistically significant difference in probability of a correct answer between 3-version voting at run-time and executing a single version.

The lack of statistically significant increase in reliability was surprising considering the claims made by the proponents of this technique so we investigated the reason for these results. Although the actual faults may have been distinct, there was a higher than random chance probability that the versions would fail on the same input cases. This type of behavior has been observed in every prior experiment involving n-version programming although the percentages have varied. It occurs either when two programmers make similar mistakes or when two dissimilar faults produce erroneous values on the same set of input data. Programmers may make similar mistakes due to specification ambiguities or due to simply finding the same parts of the problem difficult to implement correctly. In this experiment the specification was carefully reviewed and tested to remove ambiguities to the best of our abilities. It is far above the average industrial specification in quality. On the other hand, the application was quite complex, with many special cases, and occasionally programmers made similar mistakes when attempting to handle the same special case. A second question is whether the run-time voting tolerated the same faults detected by the fault elimination methods. If so, then a 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 techniques are not 100% tolerated at run-time, then it can be argued that it would be better to detect and eliminate the faults from the software prior to using it. In general, we found that the faults that were tolerated were not the same as the faults that were detected by fault elimination techniques. In fact, there was no voting triplet (of the 56 total triplets) that tolerated all the faults detected by the fault elimination techniques.

Table 3 shows the number and intersection of faults found by each technique. Note the relatively small number of faults (35) that were both tolerated at run-time and detected by a fault elimination method. Note also that fault elimination methods detected nearly three times the total number of faults tolerated by voting, and they detected more faults than voting tolerated for every version except 6 (where they were equal).

Finally, it is interesting to ask if 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 63
failure occurs (i.e., the conditional probability that a correct
lems of evaluating and keeping constant the amount of effort
This fraction was even lower, i.e., it ranged from 20.8% to
result is produced despite the failure of one of the versions).

The information in table 3 is somewhat misleading, how-
erative faults. Firm conclusions cannot be drawn from this data
given the novice nature of the participants in the fault elim-
software and all failures related to that fault eliminated. In
general, we found that even when the failure caused by a
fault is at times tolerated by a triplet, it is usually not tol-
erated every time, and there is wide variation among the
different triplets in terms of how effective they were in tol-
erating faults. 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 in that triplet causing at least one
failure. This fraction ranged from 60.4% to 88.6% with an
average of 75.9% and a standard deviation of 6.2%. That
is, even the best triplet missed 11% of the faults that it
should have been able to tolerate. 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 was even lower, i.e., it ranged from 20.8% to
61.5% with a mean of 37.9% and a standard deviation of
11.1% (see figure 1). On average the triplets only tolerated
faults 38% of the time that they caused a failure. Again this
can be explained by the large number of correlated failures
that occurred.

\[
\begin{array}{cccccccc}
\text{Version} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \text{Total} \\
\hline
\text{Tolerated by vote} & 7 & 7 & 12 & 8 & 13 & 8 & 5 & 7 & 63 \\
\text{Detected by fault elim.} & 19 & 19 & 25 & 24 & 22 & 8 & 19 & 33 & 109 \\
\text{Both vote & fault elim.} & 0 & 3 & 5 & 4 & 5 & 5 & 7 & 6 & 35 \\
\end{array}
\]

Table 3: Number of Faults Tolerated or Detected

4.2 Comparison of Fault Detection Techniques

Some comparison is possible with this data, although abso-
put into each technique. Furthermore, numbers are not re-
ally the issue; instead a more important question may be
whether different or similar faults are found by each tech-
ique. 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.

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 in-
dependent validation procedure to the outputs although there
is also a necessity to write a test harness program to imple-
ment the voting. This is not necessarily a trivial problem
when real numbers are involved since different correct re-
sults are possible from different (correct) algorithms due to
the use of limited precision arithmetic. Using a tolerance
for the comparisons will not solve the problem [6]. This has
resulted, in previous experiments in time-consuming debug-
ning of correct programs.

In discussing back-to-back testing (i.e., using voting as a
test oracle), a difficult question arises of exactly how to
define fault detection. We define a fault as detected if the
version containing it is identified as erroneous because its
answer differs from a majority of the versions. For example,
if a triplet produces Good-Good-Bad results, then the fault
responsible for producing the Bad output will be counted
as detected. However, Bad-Bad-Good (where the two Bad
results are identical) will not count as a fault detection be-
cause a fault has not been identified in the incorrect version
but instead the correct version has been erroneously iden-
tified as containing a fault. That is, a fault is counted as
detected if the voting process isolates the version that is
faulty. As another example, Bad1-Bad1-Bad2 would count
as only one fault detection (the fault responsible for the
Bad2 output) while Bad1-Bad2-Bad3 would count as three
faults detected since all three versions would need to be
debugged.

It could be argued that Bad1-Bad1-Bad2 would actually
result in finding two or three faults because in trying to
fix the Bad2 fault, the Bad1 would eventually be found.
However, we feel that once a fault is located and fixed in
the Bad version, the debuggers would probably stop. The
same type of argument could be made in the Bad-Bad-Good
case where eventually the debuggers might stumble onto the
faults causing the Bad results when they gave up attempting
to fix the Good program. This seems to us to be overly
optmistic. The tendency will be to try to get the single
answer to match the multiple answer rather than vice versa.
In fact, there were occasions when we temporarily "broke" a
correct version when trying to debug it and get it to match
the majority result. We felt that to count a fault detection
astechnique as detecting a fault, it should at least identify
the program that has failed. This decision is arbitrary but
seemed to make the most sense to the authors.

The reader should note that we are now reinterpreting
our experimental procedures differently than above. 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 method that would precede the actual produc-
tion use of the programs. There is no problem with this from a practical standpoint since the procedures are identical and differ only in the time they are performed, but it may be confusing to the reader.

Table 4 shows that there was a lot of disjunction in the faults identified by the various methods. The intersections seemed, to the authors, surprisingly small. Examination of the actual faults detected by each method lead to some generalizations about the general types of faults identified by each method.

Static data reference analysis found only uninitialized variable faults. This result follows from the definition of the technique. Three faults were found by this technique that were not detected by any other. Upon examination, it was determined that the particular 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.

Code reading found initialization faults, missing error checks (e.g., not checking for a "divide by zero" condition), and missing logic. The latter involved localized cases of missing logic only. That is, the participants did not find large global pieces of missing code or missing threads that ran through the entire program.

Voting found missing cases, but also missed large scale missing logic. Voting also detected cases of misaligned parameters and incorrect subscripts along with faults causing abends (which 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 case faults. This is consistent with past experiments, which have all reported that missing logic errors are poorly tolerated by n-version systems. Testing strategies, such as functional and structural testing, that examine special cases as well as typical cases were more successful at finding these types of faults. That is, standard testing strategies partially target precisely that portion of the problem space to which voting is known to be least effective. As discussed below, 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 (i.e., the use of voting as a test oracle is not perfect). Another unmasked fault involved the use of a wrong subscript. This is puzzling as the same thing happened in a previous experiment [7]. We cannot currently find any other explanation aside from coincidence although we are in the process of attempting to determine if an explanation exists.

Like voting, embedded assertions found misaligned parameters, incorrect subscripts, and abends. They did not detect missing cases. We are not very confident about the

2Because of the difference between the definitions of fault masking and fault detection in a voting system, the numbers in Table 4 do not correspond to the numbers in Table 3.

data for embedded 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. For example, in version 8 only a single assertion (i.e., a bounds check) was included in the code.

Functional testing identified missing case faults including large scale omissions and missing threads that were not detected by the other detection methods. It also detected faults causing abends (as did all the techniques that involved executing the programs), calculation errors (e.g., the use of the wrong variable in a computation), and missing logic to handle incorrect input data.

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 [4,5,22,24]. Of course, if one takes a larger perspective, part or all of the savings in testing may be offset by the cost of producing multiple versions of the software. However, if back-to-back testing is much more effective, then the cost arguments may be irrelevant.

There were 70 faults that were detected by the voting procedure that were not detected by any other technique. Better implementation of the other testing techniques might have found more faults (e.g., with experienced professionals instead of novices), but this would have to be proven either way. Even given the novice nature of the participants in the other testing procedures, they did find 159 faults that were not detected by the back-to-back testing. Forty one faults were detected in common. The comparison is somewhat misleading because there were faults that did not cause failures on the randomly-generated test data and that therefore could not possibly have been detected by the back-to-back testing. However, at least 8 faults caused failures during the back-to-back testing but were not identified due to identical failures in multiple versions every time they were triggered. There may have been even more that occurred, but they did not affect the output because later processing concealed the error. This is an important point and explains some of the disjunction among the faults found by the various methods. Functional and other test methods allow instrumenting the code to examine internal states, and errors were found that way that could not be identified by using only a final vote. There have been claims by advocates of n-version programming that internal checks can be made by voting on partial results. The problem is that this can only practically be accomplished at subprogram or abstract function boundaries. Otherwise, the design must be specified to the programmers and thus no diversity in design will result and fault tolerance of design errors is precluded.

Again we considered the possible variability in effectiveness between the various triplets. For each triplet, we computed the fraction of faults detected by a triplet out of the
t total number of faults that could have been detected by that triplet. This fraction varied from .905 to 1.0 with a mean of .965 and a standard deviation of .025. To determine the variability of the effectiveness of the detection among the 56 triplets, we computed the conditional probability of triplet detecting a fault given that a failure occurred as a result of that fault. This probability varied from 78.7% to 95.3% with a mean of 88.6% (see figure 2).

![Figure 2: Conditional Probability of Triplet Fault Detection](chart)

Our data suggests that using back-to-back testing on randomly-generated data is not an acceptable testing procedure by itself. 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 essentially eliminates the issue of test data generation from the issue of using the voting as a test oracle. We executed the 56 triplets on the functionally-generated test cases and did not detect any additional faults. This implies that the problem is not in the test generation method but in the identification of errors by voting.

## 5 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. There was also only one method 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. However, the results can be useful. In the few instances where there is other experimental evidence, our results tend to support and confirm previous findings. In the other cases, they represent one data point in an area of comparison where almost no experimental evidence is available. In this respect, the major contribution of our work may not be in answering questions but in determining the interesting questions to ask and in determining directions for future work.

With this perspective, the following results seem particularly interesting. Our first goal was to investigate the relationship between fault elimination techniques and software fault tolerance. We found that our data does not support the hypotheses that n-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 n-version voting did not tolerate most of the faults detected by the fault elimination techniques. We also found no significant increase in successful production of correct results using n-version programming compared with the single execution of the individual versions. Although we also found that n-version voting tolerated different faults than were detected by the fault elimination techniques, no conclusions can be drawn from this because of doubts about the ability of the novices
involved. It does, however, raise interesting questions that need to be explored.

A second goal was to investigate the use of voting in the fault elimination process. 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 to explain why these results occurred.

There are, of course, many other interesting hypotheses that can be examined using this experimental data. These will be the subject of future work.

References


