THE USE OF SELF CHECKS AND VOTING IN SOFTWARE ERROR DETECTION: AN EMPIRICAL STUDY*

Nancy G. Leveson
Stephen S. Cha

Information & Computer Science Dept.
University of California, Irvine
Irvine, CA 92717

John C. Knight

Computer Science Dept.
University of Virginia
Charlottesville, VA 22903

Timothy Shimeall

Computer Science Dept.
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents the results of an empirical study of software error detection using self checks and *n*-version voting. A total of twenty-four graduate students in computer science at the University of Virginia and the University of California, Irvine, were hired as programmers. Working independently, each first prepared a set of self checks using just the requirements specification of an aerospace application, and then each added self checks to an existing implementation of that specification. The modified programs were executed to measure the error-detection performance of the checks and to compare this with error detection using simple voting among multiple versions.

The goal of this study was to learn more about the effectiveness of such checks. The analysis of the checks revealed that there are great differences in the ability of individual programmers to design effective checks. We found that some checks that might have been effective failed to detect a fault because they were badly placed, and there were numerous instances of checks signaling non-existent errors. In general, specification-based checks alone were not as effective as combining them with code-based checks. Faults were detected by the self checks that had not

been detected previously by voting 28 versions of the program over a million randomly-generated inputs. This appeared to result from the fact that the self checks could examine the internal state of the executing program whereas voting examines only final results of computations. If internal states had to be identical in *n*-version voting systems, then there would be no reason to write multiple versions.

The programs were executed on 100,000 new randomly-generated input cases in order to compare error detection by self-checks and by *2*-version and *3*-version voting. Both self-checks and voting techniques found the same number of faults (18) for this input, although only 10 of these faults were in common, i.e., both found 8 faults that the other technique did not find. Furthermore, whereas the effective self checks detected all occurrences of errors caused by particular faults, *0fR-version voting triples and pairs were only partially effective due to correlated failures.* *n*-version voting triples and duples were only partially effective at detecting the failures caused by particular faults. Finally, checking the internal state with self-checks also resulted in finding faults that did not cause failures for the particular input cases executed. This has important implications for the use of back-to-back testing.

Index Terms: software fault tolerance, assertions, *n*-version programming, error detection, acceptance tests, software reliability.

Introduction

Crucial digital systems can fail because of faults in either software or hardware. A great deal of research in hardware design has yielded computer architectures of potentially very high reliability, such as SIFT [Wensley (1978)] and FTMP [Hopkins (1978)]. In addition, distributed systems (incorporating fail-stop processors [Schlichting and Schneider (1983)] can provide graceful degradation and safe operation even when individual computers fail or are physically damaged.

The state of the art in software development is not as advanced. Current production methods do not yield software with the required reliability for critical systems, and advanced methods of formal verification [Gries (1981)] and program synthesis [Partsch and Steinbruggen (1983)] are not yet able to deal with software of the size and complexity of many of these systems. Fault tolerance [Randell (1975)] has been proposed as a technique to allow software to cope with its own faults in a manner reminiscent of the techniques employed in hardware fault tolerance. Many detailed proposals have been made in the literature, but there is little empirical evidence to judge which techniques are most effective or even whether they can be applied successfully to real problems. This study is part of an on-going effort by the authors to collect and examine empirical data on software fault tolerance methods in order to focus future research efforts and to allow decisions to be made about real projects.

*This work was supported in part by NASA under grant numbers NAG-1-511, and NAG-1-668, by NSF CER grant DCR-8521398, and by a MICRO grant cofunded by the state of California and TRW.

Previous studies by the authors have looked at *n*-version programming in terms of independence of failures, reliability improvement, and error detection [Knight and Leveson (1986a,1986b)]. Other empirical studies of *n*-version programming have been reported [Avizienis and Kelly (1984), Bishop *et.al.* (1985), Chen and Avizienis (1978), Dunham (1986), Gmeiner and Voges (1979), Scott *et.al.* (1984)]. A study by Anderson (1985) showed promise for recovery blocks but concluded that acceptance tests are difficult to write. Acceptance tests, a subset of the more general run-time assertion or self check used in exception handling and testing schemes, are performed after completion of a program or subprogram and are essentially external checks that cannot access any local state. One of our goals was to compare the effectiveness of checks that examine only the external state with those that can check intermediate states to see if there was any advantage of one approach over the other. This type of information is needed in order to make informed choices between different types of fault tolerance techniques and to design better, more effective techniques than currently available.

In order to eliminate as many independent variables from this experiment as possible, it was decided to focus on error detection apart from other issues such as recovery. This also means that the results have implications beyond software fault tolerance alone, for example in the use of embedded assertions to detect software errors during testing [Stucki (1977)]. Furthermore, in some safety-critical systems (e.g., the Boeing 737-300 and the Airbus A310) error detection is the *only* objective. In these systems, software recovery is not attempted and, instead, a non-digital backup system such as an analog or human alternative is immediately given control in the event of a computer system failure. The results of this study may have immediate applicability in these applications. The next section describes the design of the study. Following this, the results are described for the self checks alone and then compared with the results obtained by voting.

Experimental Design

This study uses the programs developed for a previous experiment by Knight and Leveson (1986a). The main goal of the previous experiment was to investigate whether or not independently developed programs fail independently. Twenty-seven Pascal programs to read radar data and determine whether an interceptor should be launched to shoot down the object (hereafter referred to as the Launch Interceptor Program, or LIP) were prepared from a common specification by graduate students and seniors at the University of Virginia and the University of California, Irvine. Extensive efforts were made to ensure that individual students did not cooperate or exchange information about their program designs during the development phase. The twenty-seven LIP programs (along with a ‘‘gold’’ version written by the experimenters to be used as an oracle) have been analyzed by running one million randomly-generated inputs on each program and locating the individual program failures by comparing each program output with that of the gold program.

Care was taken to ensure that all faults in the programs are correctly identified [Brilliant, Knight, and Leveson (1986b)]. The gold program was carefully developed and extensively tested prior to the experiment. When the version output was different from that of the gold program, the experimenters identified the portion of the program suspected to be erroneous (i.e., the fault). They then modified the program to include a fix.

The fault in the program was considered to be correctly identified when the modified program produced the same output as the gold program. However, the gold program output was not blindly assumed to be correct. Two faults were found in the gold program during the process of fault identification.

In the present study, 8 students from UCI and 16 students from UVA were paid to instrument the programs with self-checking code in an attempt to detect errors in the programs. The participants were all paid for 40 hours of work although they spent differing amounts of time on the project. The participants were all graduate students in computer science with an average of 2.35 years of graduate study. Professional experience ranged from 0 to 9 years with an average of 1.7 years. None of the participants had prior knowledge of the LIP programs nor were they familiar with the results of the previous experiment. We found no significant correlation between the length of a participant's graduate or industrial experience and their success at writing self checks.

Eight programs were selected from the 27 and each was randomly assigned to three students (one from UCI and two from UVA). The eight programs used were randomly selected from the 14 existing programs which were known to contain two or more faults. This was done to ensure that there would be faults to detect.

Participants were provided with a brief explanation of the study along with an introduction to writing self checks. They also were provided with a chapter on error detection from a textbook on fault tolerance [Anderson and Lee (1981)]. The participants were first asked to study the LIP specification and to write checks using only the specification, the training materials, and any additional references the participants desired. When they had submitted their initial specification-based checks, they were randomly assigned a program to instrument. Self-checking code was written in Pascal, and no limitations were placed on the types of checking code that could be written except that it had to be legal Pascal.

The participants were asked to write checks with and without looking at the source code in order to determine if there was a difference in effectiveness between self checks designed by a person working from the requirements alone and those for which the person has access to and information about the program source code. It has been suggested in the literature that ideal self-checks should treat the system as a black-box and that they should be based *solely* on the specification without being influenced by the internal design and implementation [Anderson and Lee (1981)]. However, it could also be argued that looking at the code will suggest different and perhaps better ways to design self checks. Because we anticipated that the process of examining the code might result in the participants detecting faults through code reading alone, participants were asked to report any faults they thought they had detected by code reading and then to attempt to write a self check to detect the fault during execution anyway.

The participants submitted the instrumented programs along with time sheets, background profile questionnaires, and descriptions of all faults they thought they had detected through code reading. The instrumented programs were executed on the same 200 input cases that were used as an acceptability criterion in the previous experiment. The original versions were known to run correctly on that data, and we wanted to attempt to remove obvious faults introduced by the self checks. If any false alarms (i.e., faults reported that did not actually exist) were raised by these 200 inputs or if new faults were

detected that had been introduced into the program by the instrumentation, the programs were returned to the participants for correction.

After the instrumented programs had satisfied the acceptability criterion, they were executed using all the inputs on which the original programs had failed in the previous experiment. The participants were instructed to write an output message if they thought that an error had been detected by their checking code. These messages were carefully analyzed to determine their validity (i.e., error detection versus false alarm) and to identify the faults related to the errors that were detected. The same method for identifying faults was used as had been used in the original LIP program, i.e., a hypothesis about the fault was made, the code was corrected, and the program was executed again on the same input data to ensure that the output was now correct and that the self-check error message was no longer triggered. As described below, new faults (i.e., previously unidentified faults) were found by this process.

In order to compare self checks and *n*-version voting, the programs were then run on a new, randomly-generated set of 100,000 inputs. The input cases from the original experiment (that had been used up to this point) could not be used for this comparison since that data had already been determined to cause detectable failures through voting, and any comparison would be biased toward voting.

Results

In order to simulate the use of acceptance tests, participants were first asked to read through the program requirements specification and to design self checks based solely on that specification. They wrote a total of 477 checks using the specification alone. The participants were then given a particular implementation of that specification to instrument with self checks. No limitations were placed on the participants as to how much time could be spent (although they were paid only for a 40 hour week which effectively limited the amount of time spent[‡]) or how much code could be added. Table 1 describes the change in length in each program during instrumentation[†].

There is a great variation in the amount of code added, ranging from 48 lines to 835 lines. Participants added an average of 37 self checks, varying from 11 to 99. Despite this variation, we found no straight line correlation between the total number of checks inserted by a participant and the number of those checks that were effective at finding faults. That is, more checks did not necessarily mean better fault detection.

There is also no statistically significant relationship between the number of hours claimed to have been spent (as reported on the timesheets) by the participants and whether or not they detected any program faults. Figure 1 shows the amount of time each participant spent reading the specification to understand the problem, developing self checks based only on the specification, reading the source code and adding program-based self checks, and debugging the instrumented programs. Three participants (14a, 20a, and 25a) did not submit a time-sheet and are excluded from this figure.

[‡]Several reported spending more than 40 hours on the project.

[†]In order to aid the reader in referring to previously published descriptions of the faults found in the original LIP programs [Brilliant, Knight, and Leveson (1986b)], the programs are referred to in this paper by the numbers previously assigned in the original experiment. A single letter suffix is added (a, b, or c) to distinguish the three independent instrumentations of the programs.

Version #	Number of Lines				Increase		
	original	a	b	c	a	b	c
3	757	909	1152	805	152	395	48
6	643	859	887	700	216	244	57
8	600	1046	1356	824	446	756	224
12	573	1121	696	806	548	123	233
14	605	905	1342	712	300	737	107
20	533	611	1368	596	78	835	63
23	349	1065	417	544	716	68	195
25	906	1644	1016	1022	738	110	116

Table 1: Lines of Code Added During Instrumentation

Table 2 classifies the program-based self checks in terms of effectiveness by giving the number of checks for each instrumented version that were effective, ineffective, false alarms, and unknown. Checks are classified as effective if they correctly report the presence of an error stemming from a fault in the code. Two partially effective checks by participant 23a that detect an error most (but not all) of the time are counted as effective (but marked with an asterisk). All the other effective self checks written by the participants were 100% effective (i.e., they always detect an error when it occurs if they ever do).

The checks inserted by participant 23a differed from the other participants in that he used *n*-version programming to implement the self checking. That is, he rewrote the entire program using a different algorithm, and his self check became a comparison of the final results of the original program and his new program (i.e., a vote). His self check was only partially effective because of correlated failures between his version and the version he was checking. This is interesting because he had the original version and was attempting to write a completely different algorithm, i.e., he had the chance to plan diversity. This still resulted in a large percentage of correlated failures. His voting self check was effective only 16% of the time (the voting-check detected 13 failures out of a total of 80 failures that occurred). In the other 67 cases, both versions failed identically. Later in this paper we compare the effectiveness of instrumentation and *n*-version programming in general.

Ineffective checks are those that do not signal an error when one occurs during execution of the module being checked (based on the known faults in the program -- the programs were executed on data for which we already knew they would fail). False alarms signal an error when no error is present. The rest of the checks are classified as unknown because their effectiveness cannot be determined based on the input cases that have been executed to date. Executed with other data, it is possible that these checks could signal false alarms or be ineffective or effective with respect to currently unknown faults in the programs.

Version	Effectives	Ineffectives	False Alarms	Unknowns	Total
3a	1	3	0	29	33
3b	0	2	0	34	36
3c	0	3	0	11	14
6a	3	5	0	26	34
6b	0	28	1	53	82
6c	0	9	0	10	19
8a	2	0	0	13	15
8b	0	5	1	62	68
8c	1	1	3	14	19
12a	2	2	0	36	40
12b	0	5	0	17	22
12c	9	0	2	24	35
14a	0	5	0	58	63
14b	0	3	0	62	65
14c	0	1	0	16	17
20a	0	0	1	10	11
20b	2	3	2	92	99
20c	1	1	0	27	29
23a	2*	0	0	20	22
23b	0	5	0	24	29
23c	0	2	0	30	32
25a	10	0	0	30	40
25b	1	0	0	10	11
25c	0	5	0	36	41
Total	34	78	10	734	867

Table 2: Self-Check Classification

We could find no unique characteristics of the effective checks. Our problems are compounded by the fact that many of the effective checks were written for faults that were detected by the participants during code reading. Table 3 shows the faults detected by code reading along with the participant who found them. It is, of course, quite easy to write a check to detect an error caused by a known fault. In the majority of cases, the participant merely corrected the code and then compared the relevant variables at execution time. When there was a difference, the checking code wrote out an error message that an erroneous state had been detected. We separate the two cases in the rest of this section into faults detected by code reading (CR) and faults detected by code-based design checks (CD).

The ineffective self checks (i.e., checks on code that contained faults but did not detect the faults) were also examined in detail. They appear to fail due to one or more of the following reasons:

*These two checks were only partially effective.

- Wrong self-check strategy – the participant uses a type of self check inappropriate to detect the fault present in the code. The majority of the ineffective self checks failed to detect faults for this reason.
- Wrong check placement – the self check is not on the particular path in which the fault is located. Had the self check been correctly placed, it would have been effective.
- Use of the original faulty code in the self check – the participant falsely assumes a portion of the code is correct and calls that code as part of the self check.

It should be noted that the placement of the checks may be as crucial as the content. This has important implications for future research in this area and for the use of self checks in real applications.

It should not be assumed that a false alarm involved a fault in the self checks. In fact, there were cases where an error message was printed even though both the self check and the original code were correct. This was a manifestation of the Consistent Comparison Problem [Brilliant, Knight, and Leveson (1988)]. The self check made a calculation using a different algorithm than the original code. Because of the inaccuracies introduced by finite precision arithmetic compounded by the difference in order of operations, the self-check algorithm sometimes produced a result that differed from the original by more than the allowed tolerance. Increasing the tolerance does not necessarily solve this problem in a desirable way.

Participant	3a	6a	12c	20b	20c	25a
Fault	3.3	6.1, 6.2	12.1	20.2	20.2	25.1, 25.2, 25.3

Table 3: Faults Detected Through Code Reading

Table 4 summarizes the detected faults by how they were found. 19% (4 out of 21) of the detected faults were detected by specification-based checks, 43% (9 out of 21) by code reading, and 38% (8 out of 21) by code-based checks. For code-based design checks, the number of effective checks is not identical to the number of faults detected because often more than one check detected the same fault. Only 12% (4 out of 34) of the effective checks were formulated by the participants after looking at the requirements specification alone. The remaining 88% of the effective checks were designed after the participants had a chance to examine the code and write checks based on the internal state of the program.

Although it might be hypothesized that acceptance tests in the recovery block structure should be based on the specification alone in order to avoid biasing the formulator of the test, our results indicate that the effectiveness of self checks can be improved when the specification-based checks are refined and expanded by source code reading and a thorough and systematic instrumentation of the program. It appears that it is very useful for the instrumentor to actually see the code when writing self checks and to be able to examine internal states of computations in the checks.

Object	Due To			Total
	Spec-based Design (SP)	Code Reading (CR)	Code-based Design (CD)	
Faults Detected	4	9	8	21
Effective Checks	4	9	21	34

Table 4: Fault Detection Classified by Instrumentation Technique

Another way of looking at the results of this study is to consider the number of faults detected and introduced by the participants. Table 5 shows this information.

	Already Known Faults				Other Faults			Added Faults
	Present	Detected			Detected			
		SP	CR	CD	SP	CR	CD	
3a 3b 3c	4		1					
6a 6b 6c	3		2				1	1 1
8a 8b 8c	2			2			 1	 1 3
12a 12b 12c	2	1	 1				1 1	 2 2
14a 14b 14c	2							4
20a 20b 20c	2		1 1		1			1 2
23a 23b 23c	2	2						4
25a 25b 25c	3		3	 1			1	 1
total	20	3	9	3	1	0	5	22

Table 5: Summary of Fault Detection

This data makes very clear the difficulty of writing effective self checks. Of 20 previously known faults in the programs, only 12 were detected (the 15 detected known faults in Table 5 include some multiple detections of the same fault), and 6 of these were found

by code reading alone. It should be noted, however, that the versions used in the experiment are highly reliable (an average of better than 99.9% success rate on the previous one million executions), and many of the faults are quite subtle. We could find no particular types of faults that were easier to detect than others. Only 3 of the 18 detected faults were found by more than one of the three participants instrumenting the same program. Individual differences in ability appear to be important here.

One rather unusual case occurred. One of the new faults detected by participant 8c was detected quite by accident. There *is* a previously unknown fault in the program. However, the checking code contains the same fault. An error message is printed because the self-checking code uses a different algorithm than the original, and the Consistent Comparison Problem arises causing the self check to differ from the original by more than the allowed real-number tolerance. We discovered the new fault while evaluating the error messages printed, but it was entirely by chance. Erroneous triggering of self checks due to the Consistent Comparison Problem occurred in modules that did not contain a fault, and in that case the error message was classified as a false alarm (as discussed above). We were unsure whether to classify this seemingly accidental error signal as the result of an effective check or not since both the original program computation and the self check are erroneous and contain the same fault. Our decision was to classify this unusual case as an effective self check because it does signal a fault when a fault does exist, but a reasonable argument could be made for the alternative decision.

It is very interesting that the self checks detected 6 faults not previously detected by comparison of twenty-eight versions of the program over a million inputs. The fact that the self checks uncovered new faults even though the programs were run on the same inputs that did not reveal the faults through voting implies that self checking may have advantages over voting alone. To understand why, it is instructive to examine an example of one of the previously undetected faults.

Some algorithms are unstable under a few conditions. More specifically, several mathematically valid formulae to compute the area of a triangle are not equally reliable when implemented using finite precision arithmetic. In particular, the use of Heron's formula:

$$area = \sqrt{s * (s - a) * (s - b) * (s - c)}$$

where a, b, and c are the distances between the three points and s is $(a + b + c)/2$, fails in the rare case when all the following conditions are met simultaneously:

- Three points are almost co-linear (but not exactly). s will then be extremely close to one of the distances, say a, so that $(s - a)$ will be very small. The computer value of $(s - a)$ will then be of relatively poor accuracy because of round-off errors (around 10^{-16} in the hardware employed in this experiment).
- The product of the rest of the terms, $s * (s - b) * (s - c)$, is large enough (approximately 10^{4}) to make rounding errors significant through multiplication (approximately 10^{-12}).
- The area formed by taking the square root is slightly larger than the real number comparison tolerance (10^{-6} in our example) so that the area is not considered zero.

Other formulas, for example

$$area = \frac{x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1}{2}$$

where x_i and y_i are the coordinates of the three points, did not fail because the potential roundoff errors cannot become ‘‘significant’’ due to the order of operations. Two of the six previously unknown faults detected involved the use of Heron’s formula. Because the source of the unreliability is in the order of computation and inherent in the formula, relaxing the real number comparison tolerance will not prevent this problem. The fault in Heron’s formula was not detected during the previous executions because the voting procedure compared the final result *only*, whereas the self check verified the validity of the intermediate results as well. For the few cases in which it arose, the faults did not affect the correctness of the final output. However, under different circumstances the final output would have been incorrect. We did not include in this analysis any fault for which we could not find legitimate inputs that would have made the programs fail.

Although new faults were introduced through the self checks, this is not very surprising. It is known that changing someone else’s program is difficult and whenever new code is added to a program there is a possibility of introducing faults. All software fault tolerance methods involve adding additional code of one kind or another to the basic application program. The major causes of the new faults were an algorithmic error in a redundant computation, use of an uninitialized variable during instrumentation, a logic error, use of Heron’s formula, infinite loops added during instrumentation, an out of bounds array reference, etc.

The use of uninitialized variables occurred due to incomplete program instrumentation. A participant would declare a temporary variable to hold an intermediate value during the computation, but fail to assign a value on some path through the computation. A more rigorous testing procedure for the assertions may have detected these faults earlier. The instrumented versions were not run on many test cases before being evaluated. In most realistic situations, assertions or self checks would be added before rigorous testing of the program was performed instead of afterward as in our case. It is significant, however, that many of the same types of faults were introduced during instrumentation as were added during the original coding and thus presumably might also escape detection during testing in the same way the other ones did.

Comparisons with Voting

When making decisions about what type of error detection scheme to use, it is important to have some comparison data. In this study, we compared the error detection effectiveness of general self checks and simple voting schemes. The faults detected through code reading are counted as detected by self checks; code reading is an integral part of the process of instrumenting programs. The faults detected through code reading would probably be fixed in a realistic setting (instead of writing a check for it), but the fault has still been detected. In order to avoid confusion, we refer to the entire process as instrumentation of the code. Also, because the self-check effectiveness had been investigated using the inputs on which we knew that voting detected failures, we ran the programs using both self checks and voting on an additional 100,000 randomly-generated inputs and did the comparisons only for these new inputs.

Care must be taken in evaluating the resulting data since this was not part of the original experimental design, and the experiment as designed has limitations in the comparability of the data. For example, there are 3 instrumentations of each version whereas there are 21 different voting triples and 7 voting pairs that could possibly detect the faults in a particular version. Furthermore, as discussed below, ‘‘coverage’’ (the probability of detecting a fault given that it produces an error and that there is a check in place for that error) is difficult to factor out so that comparing the absolute number of faults detected is somewhat misleading. However, the data does suggest some hypotheses that might be explored either theoretically or through further controlled empirical evaluation.

There are many types of voting schemes possible and different decisions will affect the outcome. We considered the alternatives and selected the one that appeared most reasonable to us. The original LIP experiment [Knight and Leveson (1986a)] used vector voting. Each program produces a 15 by 15 Boolean array, a 15 element Boolean vector, and a single Boolean launch condition (a total of 241 outputs) for each set of inputs. In vector voting, an error is detected if any of the 241 results differ between the versions. For example, if three versions provide the three results 110, 011, and 111 (where 1 stands for a correct partial result and 0 stands for an incorrect partial result), this is counted as three different answers and the triplet fails with no answer.

It has been suggested that bit-by-bit voting, where the answer is formed by determining the majority of each individual result, would have improved the voting system reliability of our programs [Kelly *et.al.* (1986)]. In the above example, each of the three partial results has a majority of correct answers so the final answer would be correct. In bit-by-bit 3-way voting, it is impossible for the voting system to fail to produce an answer, i.e., a majority always exists among three Boolean results. This is possible, obviously, only because of the Boolean nature of each launch condition. The launch conditions were of the sort: ‘‘There exists at least one set of three consecutive data points that cannot all be contained within or on a circle of radius X.’’

In order to determine whether there was a significant difference between vector voting and bit-by-bit voting for our problem, we performed each and compared the results. There were 100,000 input cases executed, and there are 56 possible *3*-version combinations of 8 programs. Therefore, a total of 5,600,000 votes were taken. The results are shown in Table 6 where a result is classified as Wrong if a majority of the three versions agree on a wrong answer and it is classified as No Answer if the three versions all disagree.

	Bit-by-Bit		Vector	
	Count	Percent	Count	Percent
Correct	5599414	99.9895	5599414	99.9895
Wrong	586	0.0105	562	0.0100
No Answer	0	0.0	24	0.0004

Table 6. Three-Version Voting

For this data there is no difference in the resulting probability of correct answer between bit-by-bit and vector voting. The 24 cases where there had previously been no agreement

all became wrong answers using bit-by-bit voting. Vector voting is safer because it is less likely to allow a wrong answer to go undetected at execution time (no answer is usually safer than a wrong answer), and we use majority vector voting for the rest of this paper.

There were 2,800,000 *2*-version votes (28 combinations of 2 programs executing 100,000 input cases). Table 7 shows the results where a wrong answer results if both versions are identically wrong and no answer results if the two versions disagree[†]. Wrong answers are identified by using *9*-version voting (the 8 versions plus the gold version). Faults that cause common failures in all 9 versions will, of course, not have been detected. As expected, the probability of a correct answer for *2*-version voting is lower than for either a *3*-version system or a single version run alone (see Table 8).

	Count	Percent
Correct	2796359	99.8700
Wrong	112	0.0040
No Answer	3529	0.1260

Table 7. Two-Version Voting

The probability of producing a correct answer for the individual programs is shown in the Table 8 below. The mean success probability for individual programs is 99.933%.

Version	Cases Failed	Success Percentage
3	223	99.777
6	61	99.939
8	25	99.975
12	49	99.951
14	140	99.860
20	25	99.975
23	4	99.996
25	10	99.990

Table 8. Individual Version Performance

Table 9 contains the comparison data for fault detection using voting and instrumentation[‡]. The previously unknown faults detected by instrumentation are labelled 6.4, 8.3, 12.3, 12.4, 20.3, and 25.4.

[†]In 6 out of the 3529 No Answer cases in Table 7, the versions returned two distinct answers which were both wrong.

[‡]The data here differs slightly from that in Table 5 because a different set of input cases was used.

Faults	Voting	Instrumentation
3.1	√	no
3.2	yes	no
3.3	yes	yes
3.4	yes	no
6.1	yes	yes
6.2	yes	yes
6.3	yes	no
6.4	no	yes
8.1	yes	yes
8.2	yes	yes
8.3	no	yes
12.1	yes	yes
12.2	yes	no
12.3	no	yes
12.4	no	yes
14.1	yes	no
14.2	yes	no
20.1	yes	no
20.2	no	yes
20.3	no	yes
23.1	yes	yes
23.2	yes	yes
25.1	no	yes
25.2	yes	yes
25.3	yes	yes
25.4	no	yes
total	18	18

Table 9: Comparison of Fault Detection

In Table 9, a fault is counted as detected by voting if an error caused by the fault is detected at least once (but not necessarily every time it occurs) and by at least one of the voting triples or duples (but not necessarily by all of them). In fact, as discussed below, voting was only partially effective at detecting and tolerating errors for the majority of faults and for the majority of triples and duples. The instrumentation is considered to have detected a fault if at least one of the three instrumentations detected an error resulting from the fault. With this definition of fault detection, voting detected 8 faults that were not detected through instrumentation, the instrumentation detected 8 faults that were not detected by voting, and 10 faults were detected by both. Voting and instrumentation detected the same number of faults in total.

Another type of comparison is to consider the effectiveness or ‘‘coverage’’ of the technique in detecting faults in terms of how often a fault was detected given that it caused an error and a potentially effective check was made. Tables 10a and 10b show the rates of detection for each of the voting triples and duples. The only faults included in

the tables are those that were detected by voting on the original one million input cases; the additional faults detected by assertions alone are not included as they would not generate any entries in the table. The second row shows the number of failures caused by each fault for the 100,000 input cases executed. Each row in the table below this row then shows the number of times the triple detected these failures. Two faults (i.e., 20.2 and 25.1) that caused failures on the original 1,000,000 input cases did not cause failures for these 100,000 input cases (although the assertions detected them because they did cause errors in the internal state of the program). A dot in the table means that that position is irrelevant, i.e., the fault was in a program that was not one of the members of the triple. An asterisk next to a number indicates that not all of the failures were detected. The bottom line gives the percentage of time any triple or duple detected that particular fault given that it caused an error. For triples, this ranged from 29% to 100%, with an average of 0.68 and a standard deviation of 0.32. For *2*-version voting, the results are a little better (as would be expected) but the coverage still averaged only 0.82 (standard deviation of 0.19) with several versions as low as 0.57.

This contrasts with instrumented self checks where if there was a check in place that ever detected an error caused by a fault, then it *always* detected the errors caused by that fault. This was true for both the one hundred thousand input cases executed in this part of the experiment and for the previous one million test cases except for version 23a where the participant used *n*-version voting to implement the self checking as discussed earlier. For these particular 100,000 input cases, version 23 failed only twice and both were detected by the voting-check inserted by participant 23a. However, on the previous one million input cases, the voting-check by 23a was effective for only 13 out of 80 failures.

It appears that the self checks are highly effective because they check the internal state and, therefore, consistently find the errors they are capable of detecting. The voting procedure only checks the results of computations and not the internal consistency of the intermediate results and other parts of the internal state. Therefore voting is subject to correlated failures in the multiple versions.

Using cross-check points in voting to compare the results of computations internal to the program (and not just the final output) cannot be used to solve the problem as long as truly diverse algorithms are used in the independent versions. Diversity implies that the internal states of the programs will not be identical. If they are identical, then there is no diversity and no potential fault tolerance. Some comparison of intermediate results is, of course, possible, but the only way to guarantee this comparability is to decrease the diversity by requiring the programmers to use similar designs, variables, and algorithms. At the extreme, this results in totally specified and thus identical versions.

On the other hand, placement of the self checks is critical (as noted above) because they are not placed just at the end or in synchronized locations, as in voting, where all paths are guaranteed to reach them. So potentially effective checks may be bypassed. Furthermore, although the effective checks were 100% effective, only 3 of the 18 faults found were detected by more than one of the three instrumentations that could have found it. Of course, more than one person could instrument the same program. From our data, it appears that examining a team approach to instrumentation would be worthwhile.

Another interesting thing to notice is that faults 20.2 and 25.1 did not cause failure during execution of any of the 100,000 input cases (although they did on the previous

million input cases). The self checks detected these faults (and six others that voting did not detect even on the million input cases) due to the fact that they could check the internal state of intermediate computations. Voting could not detect these faults for this particular input data because there were no erroneous outputs. In a testing environment, self checks may find faults that back-to-back testing (i.e., using the comparison of the results of multiple versions as a test oracle) [Bishop *et.al.* (1985), Brilliant (1987), Ramamoorthy *et.al.* (1981), Saglietti and Ehrenberger (1986)] does not find. This is consistent with our results for another empirical study that included a comparison of standard testing methods and back-to-back testing [Shimeall and Leveson (1988)].

Conclusions

Almost no empirical data exists on the effectiveness of using self-checks to detect errors at run-time, and no previous studies have compared error detection using*n*-version voting with self-checks. Several interesting results were obtained from this study that should guide us and others in the evaluation of current proposals for fault tolerance and error detection, in the design of new methodologies, and in the design of further experiments.

The first goal of this experiment was to instrument the programs with self-checks and determine how effective these checks were in detecting errors when the programs were run on data that was known to make them fail. We found that detecting errors is quite difficult in programs whose reliability is already relatively high and the faults very subtle. Out of a possible 60 known faults that could have been detected, the participants detected only 6 by specification-based and code-based checks and another 9 by code reading while writing the code-based checks. Only 11 of the 24 participants wrote checks that detected faults and there was little overlap in the faults detected which implies that there are great individual differences in the ability of individuals to design effective checks. This suggests that more training or experience might be helpful. Our participants had little of either although all were familiar with the use of pre- and post-conditions and assertions to formally verify programs. The data suggests that it might also be interesting to investigate the use of teams to instrument code.

Placement of self checks appeared to cause problems. Some checks that might have been effective failed to detect a fault because they were badly placed. This implies either a need for better decision-making and rules for placing checks or perhaps different software design techniques to make placement easier.

Surprisingly, the self-checks detected 6 previously unknown faults that had not been detected by *28*-version voting on one million randomly-generated input cases. This should give pause to those with high confidence that all faults have been eliminated from a complex program. The fact that the self checks uncovered new faults that were not detected by voting on the same input cases implies that self-checking may have important advantages over voting. In particular, comparing only the final results of a program or even the final results of a computation within a program may be less effective in finding errors than verifying the validity of the intermediate results and structures of the program. That is, self-checks allow verification of more than just the final results of computations.

Specification-based checks alone were not as effective as using them together with code-based checks. This again was surprising as it conflicts with previous hypotheses, and it implies that fault tolerance may be enhanced if the alternate blocks in a recovery block scheme, for example, are also augmented with self checks along with the usual acceptance test. This appears to be true also for pure voting systems. A combination of fault-tolerance techniques may be more effective than any one alone. More information is needed on how best to integrate these different proposals. In most situations, it will be impractical to attempt to completely implement multiple fault tolerance schemes given the relatively large cost of most of these techniques. Therefore, there needs to be some determination of what are the most cost/effective techniques to use.

The comparison data between self checking and voting needs to be treated with some care. However, the results are interesting and suggest that further study might be fruitful. Although there were only three attempts to write self checks to detect a particular fault compared to the 21 voting triples and 7 voting duples that could possibly find each fault, self checking found as many faults as voting. When comparing coverage, self checks were much more effective at finding errors given that the error occurred and a potentially effective check was in place for it. For our data, effective self checks were 100% effective whereas voting was found to be only partially effective a large percentage of the time. The same number of faults were detected by each of these techniques that were not detected by the other implying that they are not substitutes for each other.

Finally, faults were detected by self-checks that did not cause failures for the individual versions (and thus were not detected by voting even though two had been detected by voting on different input data). This has important implications for testing. Back-to-back testing has been suggested as a method for executing large amounts of test data by using voting as the test oracle. However, our data implies that back-to-back testing alone may not find the same faults as other types of testing that involve instrumenting the code with checks on the internal state.

Further empirical studies and experiments are needed before it will be possible to make informed choices among fault detection techniques. Very little empirical evidence is available. This experiment, besides substantiating some anticipated results and casting doubt on some previously-suggested hypotheses, provides information that can help to focus future efforts to improve fault tolerance and error detection techniques and to design future experiments. Potentially useful future directions include the following:

[1] The programs were instrumented with self checks in our study by participants who did not write the original code. It would be interesting to compare this with instrumentation by the original programmer. A reasonable argument could be made both ways. The original programmer, who presumably understands the code better, might introduce fewer new faults and might be better able to place the checks. On the other hand, separate instrumentors might be more likely to detect faults since they provide a new view of the problem. More comparative data is needed here. It is interesting that the original programmers, in a questionnaire they submitted with their programs, were asked what was the probability of residual errors in their programs, and if there were errors, what parts of the program might contain them. Most were confident that there were no residual errors and were almost always wrong when guessing about where any errors might be located.

[2] Another interesting question is whether the effectiveness of the code reading was influenced by the fact that the participants read the code with the goal of writing self checks. It would be interesting to compare this with more standard code reading strategies.

[3] The process of writing self checks is obviously difficult. However, there may be ways to provide help with this process. For example, Leveson and Shimeall (1983) suggest that safety analysis using software fault trees [Leveson and Harvey (1983)] can be used to determine the content and the placement of the most important self checks. Other types of application or program analysis may also be of assistance including deriving the assertions or self-checks from formal specifications. Finally, empirical data about common fault types may be important in learning how to instrument code with self checks. All of these different strategies need to be experimentally validated and compared.

Acknowledgements

The authors are pleased to acknowledge the efforts of the experiment participants: David W. Aha, Tom Bair, Jack Beusmans, Bryan Catron, Harry S. Delugach, Siamak Emadi, Lori Fitch, W. Andrew Frye, Joe Gresh, Randy Jones, James R. Kipps, Faith Leifman, Costa Livadas, Jerry Marco, David A. Montuori, John Palesis, Nancy Pomicter, Mary Theresa Roberson, Karen Ruhleder, Brenda Gates Spielman, Yellamraju Venkata Srinivas, Tim Strayer, Gerald Reed Taylor III, and Raymond R. Wagner, Jr.

References

1. T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, ‘‘An Evaluation of Software Fault Tolerance in a Practical System’’, *Digest of Papers FTCS-15: Fifteenth Annual Symposium on Fault-Tolerant Computing,*, Ann Arbor, Michigan, pp. 140-145, June 1985.

2. T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice* Englewood Cliffs, NJ, Prentice-Hall Intl., 1981.

3. D.M. Andrews and J.T. Benson, ‘‘An Automated Program Testing Methodology and its Implementation,’’ *Proc. 5th Int. Conference on Software Engineering*, San Diego, CA, pp. 254-261, March 1981.

4. A. Avizienis and L. Chen, ‘‘On the Implementation of *N*-version Programming for Software Fault-Tolerance During Execution’’, *Proceedings of COMPSAC 77* pp. 149-155, November 1977.

5. A. Avizienis and J.P.J. Kelly, ‘‘Fault Tolerance By Design Diversity: Concepts and Experiments’’, *IEEE Computer Magazine* , Vol. 17, No. 8, pp. 67-80, August 1984.

6. P. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti, and S. Yoshimura, ‘‘PODS - A Project on Diverse Software’’, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, pp. 929-940, September 1986.

7. S.S. Brilliant, ‘‘Testing Software Using Multiple Versions,’’ Ph.D. Dissertation, University of Virginia, September 1987.

8. S.S. Brilliant, J.C. Knight, and N.G. Leveson, ‘‘The Consistent Comparison Problem in *N*-Version Software’’, *IEEE Trans. on Software Engineering*, 1989 (in press).

9. S.S. Brilliant, J.C. Knight, and N.G. Leveson, ‘‘Analysis of Faults in an *N*-Version Software Experiment’’, submitted for publication, 1986b.

10. L. Chen and A. Avizienis, ‘‘*N*-version programming: A fault-tolerance approach to reliability of software operation,’’ *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.

11. J.R. Dunham, ‘‘Software Errors in Experimental Systems Having Ultra-Reliability Requirements’’, *Digest of Papers FTCS-16: Sixteenth Annual Symposium on Fault-Tolerant Computing,*, Vienna, Austria, pp 158-164, July 1986

12. L. Gmeiner and U. Voges, ‘‘Software Diversity in Reactor Protection System: An Experiment’’, *Proceedings of IFAC Workshop SAFECOMP ’79* pp 75-79, 1979

13. D. Gries, *The Science Of Programming,* Springer Verlag, 1981.

14. A.L. Hopkins, et al., ‘‘FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft’’, *Proceedings of the IEEE,* Vol. 66, pp. 1221-1239, October 1978.

15. J.P.J. Kelly, et.al., ‘‘Multi-Version Software Development,’’ *Proceedings of IFAC Workshop Safecomp ’86,* Sarlat, France, pp. 43-49, October 1986.

16. J.C. Knight and N.G. Leveson, ‘‘An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming’’, *IEEE Transaction on Software Engineering,* pp. 96-109, January 1986a.

17. J.C. Knight and N.G. Leveson, ‘‘An Empirical Study of Failure Probabilities in Multi-Version Software’’, *Digest of Papers FTCS-16: Sixteenth Annual Symposium on Fault-Tolerant Computing,,* Vienna, Austria, pp.165-170, July 1986b.

18. N.G. Leveson, and P.R. Harvey, ‘‘Analyzing Software Safety’’, *IEEE Transactions on Software Engineering,* Vol. SE-9, No. 5, pp 569-579, September, 1983.

19. N.G. Leveson, and T.J. Shimeall, ‘‘Safety Assertions for Process-Control Systems’’, *Digest of Papers FTCS-13: Thirteenth Annual Symposium on Fault-Tolerant Computing,* Milan, Italy, pp 236-240, June 1983.

20. H. Partsch and R. Steinbruggen, ‘‘Program Transformation Systems’’, *ACM Computing Surveys,* Vol. 15, No. 3, pp. 199-236, September 1983.

21. C.V. Ramamoorthy, Y.K Mok, E.B. Bastani, G.H. Chin, 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.

22. B. Randell, ‘‘System Structure for Software Fault-Tolerance,’’ *IEEE Transactions on Software Engineering,* Vol. SE-1, No. 2, pp. 220-232, June 1975.

23. F. Saglietti and W. Ehrenberger, ‘‘Software Diversity — Some Considerations about its Benefits and its Limitations,’’ *Safecomp ’86,* Sarlat, France, pp. 27-34, October 1986.

24. R.D. Schlichting and F.B. Schneider, ‘‘Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems’’, *ACM Transactions On Computer Systems,* Vol. 1, pp. 222-238, August 1983.

25. K.R. Scott, J.W. Gault, D.F. McAllister, and J. Wiggs, ‘‘Experimental Validation of six Fault Tolerant Software Reliability Models’’, *Digest of Papers FTCS-14:*

Fourteenth Annual Symposium on Fault-Tolerant Computing, Kissemmee, NY, pp 102-107, 1984.

26. T.J. Shimeall and N.G. Leveson, ‘‘An Empirical Comparison of Software Fault Tolerance and Fault Elimination,’’ *Proc. 2nd Workshop on Software Testing, Verification, and Analysis,* Banff, pp. 180-187, July 1988.

27. L.G. Stucki, ‘‘New Directions in Automated Tools for Improving Software Quality’’, *Current Trends in Programming Methodology - Volume II: Program Validation, Prentice Hall, pp. 80-111, 1977*

28. J.H. Wensley, et al., ‘‘SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control’’, *Proceedings of the IEEE,* Vol. 66, pp. 1240-1254, October 1978.