The Use of Self Checks and Voting in Software Error Detection: An Empirical Study

NANCY G. LEVESON, STEPHEN S. CHA, JOHN C. KNIGHT, AND TIMOTHY J. SHIMEALL

Abstract—**This paper presents the results of an empirical study of software error detection using self checks and *N*-version voting. A total of 24 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 an error because they were badly placed, and there were numerous instances of checks signaling nonexistent errors. In general, specification-based checks alone were not as effective as combining them with code-based checks. Using self checks, faults were identified 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 led to the identification of the same number of faults for this input, although the identified faults were not the same. Furthermore, whereas the self checks were always effective at detecting an error caused by a particular fault (if they ever did), *N*-version voting triples and pairs 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—**Acceptance tests, assertions, error detection, *N*-version programming, software fault tolerance, software reliability.**

I. 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 [29] and FTMP [14]. In addition, distributed systems (incorporating fail-stop processors [24]) 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 [13] and program synthesis [20] are not yet able to deal with software of the size and complexity of many of these systems. Fault tolerance [22] 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.

Previous studies by the authors have looked at *N*-version programming in terms of independence of failures, reliability improvement, and error detection [16], [17]. Other empirical studies of *N*-version programming have been reported [4], [6], [10], [11], [12], [25]. A study by Anderson *et al.* [1] 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 evaluated after execution 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

Manuscript received August 22, 1988; revised November 8, 1989. Recommended by Y. Matsumoto. This work was supported in part by NASA under Grants NAG-1-511 and NAG-1-668, by the National Science Foundation under CER Grant DCR-8521398, and by a MICRO grant cofunded by the state of California and TRW.

N. G. Leveson and S. S. Cha are with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

J. C. Knight is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903.

T. J. Shimeall is with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943.

IEEE Log Number 8933743.

use of embedded assertions to detect software errors during testing [3], [5], [28]. Furthermore, in some safety-critical systems (e.g., avionics systems in 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 nondigital 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.

II. Experimental Design

This study uses the programs developed for a previous experiment by Knight and Leveson [16]. In the previous experiment, 27 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 27 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 the detected faults in the programs are correctly identified [8]. The gold program was 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 correct the suspected fault. The fault 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. Although neither self checks nor voting can actually detect a fault (i.e., they detect the errors and failures caused by the fault), in this paper we loosely say that a technique has "detected a fault" if it detects an error that was caused by that fault. Obviously, fault detection itself requires human intervention.

In the present study, 8 students from UCI and 16 students from UVA were hired to instrument the programs with self-checking code in an attempt to detect errors. The participants were all employed for 40 hours 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 LIP versions, 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 that 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 [2]. In this chapter, the general concept of self checking is described, and checks are classified into the following categories:

1) *Replication*—checks that involve some replication of an activity and comparison of outputs.

2) *Timing*—checks that determine whether the operation of a component meets timing constraints.

3) *Reversal*—checks that take the output(s) from a system and calculate what the input(s) should have been in order to produce those output(s).

4) *Coding*—checks that maintain redundant data in the representation of an object or set of objects in some fixed relationship with the (nonredundant) data and ensure that the relationship holds.

5) *Reasonableness*—checks that determine whether the states of various (abstract) objects in the system are "reasonable" given the intended usage and purpose of those objects using knowledge of the system's design.

6) *Structural*—checks on the semantic and structural integrity of data structures.

7) *Diagnostic*—checks on the performance of the components from which the system is constructed rather than checks on the behavior of the system itself. Such checks involve exercising a component with a set of inputs for which the correct outputs are known.

We did not require that the participants use these or any other particular types of checks because we had no *a priori* knowledge of which checks might be relevant to the software at hand and because we wanted to allow the maximum flexibility and opportunity for creativity on the part of the participants. To the best of our knowledge, this was the first experiment of its kind and we did not want to restrict the participants in any way since any restrictions might limit the effectiveness of the checks unintentionally. Our goal was to determine whether self checks, in general, can be an effective means of error detection, to provide some preliminary comparisons with simple voting schemes, and to gather information that could be used to design and focus future experiments both by ourselves and others. Often the most important information derived from the early experiments in a particular area is to determine what are the most important variables on which to focus in future studies. Overconstraining the first empirical

studies can mean that important information is inadvertently missed.

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.

The participants were asked to write checks first without access to the source code and then with the source code available 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 designed with knowledge of 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 [2]. 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 detected by code reading and then to attempt to write a self check to detect errors caused by these faults 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 acceptance procedure in the previous experiment. The original versions were known to run correctly on that data, and we wanted to remove obvious faults introduced by the self checks. If during the acceptance procedure a program raised a false alarm, i.e., reported an error that did not exist, or if new faults were detected in the instrumentation itself, the program was returned to the participant for correction.

After the instrumented programs had satisfied the acceptance procedure, 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 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.

In order to compare self checks and N-version voting (i.e., detecting errors through voting by N versions on the results and taking the majority result as correct), the instrumented 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.

III. Results

The results are presented in two parts. The first part looks at the effectiveness of self checks in detecting errors. The second part compares self checks and voting in terms of fault detection.

A. Error Detection Using Self Checks

During the first phase of the experiment in which self checks were based on the program requirements specification alone, a total of 477 checks were written. The participants were then given a particular implementation of that specification, i.e., one of the programs, to instrument with self checks. Each of the eight programs were given to three participants so that individual differences could be partially factored out of the experiment. 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[1]) or how much code could be added. Table I shows the change in length in each program during instrumentation. In order to aid the reader in referring to previously published descriptions of the faults found in the original LIP programs [8], 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.

There was a great deal of 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 obvious correlation between the total number of checks inserted by a participant and the number of those checks that were effective at finding errors. That is, more checks did not necessarily mean better error detection.

There is also no 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 errors. Fig. 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.[2]

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

[2]Three participants (14a, 20a, and 25a) did not submit a timesheet and are excluded from Fig. 1.

TABLE I
LINES OF CODE ADDED DURING INSTRUMENTATION

Version #	Number of Lines original	a	b	c	Increase 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

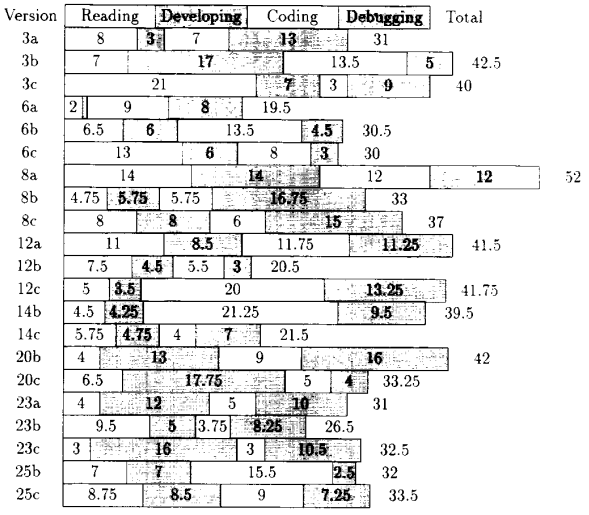


Fig. 1. Summary of participant timesheets.

Table II presents the performance of the program-based self checks. For each instrumented version, the number of checks that were effective, ineffective, false alarms, and unknown are shown.

Checks were classified as effective if they correctly report the presence of an error stemming from a fault in the code. Ineffective checks are those that did not signal an error when one occurred during execution of the module being checked (based on the known faults in the program and using data for which we knew already that 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.

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 other self checks that we classified as effective always detected an error if it occurred. 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

TABLE II
SELF-CHECK CLASSIFICATION

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

*These two checks were only partially effective.

results of the original program and his new program. This approach was not expected but was not precluded by the instructions given to the participants. His self check was only partially effective because of coincident failures between his version and the version he was checking. This is significant 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, i.e., the voting check detected 13 failures out of a total of 80 failures that occurred. In the other 67 cases, both versions failed identically.

We could find no unique characteristics of the effective checks. As explained in the previous section, we purposely did not restrict the participants to any particular types of self checks, and we found we were unable to classify the self checks actually used. Classification of checks is, in general, extremely imprecise, which makes enforcement of the required use of a particular type of check virtually impossible and precludes any meaningful postexperiment classification by us. As an example of the difficulty, consider a check that computes an approximation of a required result. If the approximation is coarse, this might be classified either as a replication check or as a reasonableness check. Similarly, one might classify branch checks as a separate type of check or they could be considered as a type of consistency check. If branch checks attempt to detect errors through redundant computation of the branch conditions, they could even be classified as replication checks. Our attempts to classify the more than 800 checks led to hopeless inconsistency among the classifiers.

Our problems are compounded by the fact that some of

the effective checks were written for faults that were detected by the participants during code reading. Table III shows the fault identifiers for those faults found 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 through code reading (CR) and faults detected through code-based design checks (CD).

Even if we could have classified the checks, it would be misleading to attempt to draw conclusions from this data about the types of checks that are effective since the checks were in different programs with different types of errors. It would be inappropriate (and even misleading) to claim that checks of one type are more effective than others based on our data. The types of effective checks for a particular program probably will vary widely depending on the type of application and the type of errors in the program. For example, if the program is computation-intensive and if there exist several algorithms to compute the answer, replication checks might be effective. If, on the other hand, the application involves the manipulation of complicated data structures using relatively simple computations, structural checks might be the most effective. The algorithms and errors were different in the different LIP programs, and therefore the effective checks would also be expected to be different.

We examined the ineffective self checks to determine why they did not signal errors when one occurred during execution of the module being checked. They appear to be ineffective for one or more of the following reasons:

- *Wrong self-check strategy*—The participant failed to check the erroneous variables or checked them for properties unrelated to the error. The majority of the ineffective self checks failed to detect errors 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 [9]. 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.

TABLE III
FAULTS DETECTED THROUGH CODE READING

Version	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 IV summarizes the detected faults by how they were found. Nineteen % of the detected faults were detected by specification-based checks, 43% by code reading, and 38% 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 errors caused by 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 has been 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.

Previously unknown faults were detected during instrumentation, and new faults were added. Table V presents, for each participant, the number of known and previously-unknown faults that were detected and the number of new faults that were introduced. 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 V include some double 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

TABLE IV
FAULT DETECTION CLASSIFIED BY INSTRUMENTATION TECHNIQUE

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 V
SUMMARY OF FAULT DETECTION

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

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. We were unsure whether to classify this seemingly accidental error signal as the result of an effective check since both the original program computation and the self check are erroneous and contain the same fault. We decided to classify this unusual case as an effective self check because it does signal an error when a fault does exist.

It should be noted that the self checks detected 6 faults not previously detected by comparison of 28 versions of the program over a million inputs. The fact that the self checks uncovered errors caused by new faults even though the programs were run on the same inputs that did not reveal the errors through voting implies that self checking may have advantages over voting alone. A detailed examination of one of the previously undetected faults helps to illustrate why this is true.

Some algorithms are unstable under a few conditions. More specifically, several mathematically valid formulas 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:

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

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

- The three points are almost collinear (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 roundoff 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

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

where x_i and y_i are the coordinates of the three points, provide the correct answer under these circumstances; 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 faults involving the use of Heron's formula were not detected during the previous executions because the voting procedure compared the final result *only*, whereas the self checks verified the validity of the intermediate results as well. For the few input cases in which it arose, the faults did not affect the correctness of the final output. However, under different circumstances (i.e., different inputs) the final output would have been incorrect. We did not categorize anything as a fault unless we could find legitimate inputs that will make the programs fail due to this fault.

Although new faults were introduced through the self checks, this is not surprising. It is known that changing someone else's program is difficult: 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. Examples of new faults introduced by the self checks are the use of an uninitialized variable during instrumentation, algorithmic errors in computations, 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 self checks 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 some presumably might also escape detection during testing.

B. Comparisons with Voting

When making a decision about which type of error detection scheme to use, it is important to have 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. 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 executed 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.

There are many different voting schemes and which is used may affect the outcome. In the original LIP experiment [16], each program produced 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, and vector voting was used. 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 100, 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 [15]. In the example above, 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 the data.

To determine whether there was a significant difference between vector voting and bit-by-bit voting for our application, we performed each and compared the results. One hundred thousand input cases were executed, and voting was performed on the results for each of the 56 possible 3-version combinations of the 8 programs. Therefore, a total of 5,600,000 votes were taken.

The results are shown in Table VI where a result was classified as Wrong if a majority of the three versions agreed on a wrong answer, and it was classified as No Answer if there was no majority. 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 (i.e., some systems may have fail-safe mechanisms to recover when the computer fails to produce a result), and we use majority vector voting for the rest of this paper.

TABLE VI
THREE-VERSION VOTING

	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

A total of 2 800 000 2-version votes were taken (28 combinations of 2 programs executing 100 000 input cases), and the results are shown in Table VII. A wrong answer was recorded if both versions were identically wrong, and no answer was recorded if the two versions disagreed.[3] Wrong answers were 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 below).

The probability of producing a correct answer for the individual programs is shown in Table VIII. The average probability of success for individual programs is 99.933%.

Table IX contains the comparison data for fault detection using voting and instrumentation.[4] The previously unknown faults detected by instrumentation are labeled 6.4, 8.3, 12.3, 12.4, 20.3, and 25.4. In Table IX, a fault is counted as detected by voting if an erroneous result caused by that fault is detected at least once (but not necessarily every time it occurs) and by at least one of the voting triples or pairs (but not necessarily by all of them). In fact, as discussed below, voting was only partially effective at detecting (and tolerating) erroneous results for the majority of faults and for the majority of triples and pairs. 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 led to the detection of 8 faults that were not detected through instrumentation, the instrumentation led to the detection of 8 faults that were not detected by voting, and 10 faults were detected by both.

[3] In 6 out of the 3529 No Answer cases in Table VII, the versions returned two distinct wrong answers.

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

TABLE VII
TWO-VERSION VOTING

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

TABLE VIII
INDIVIDUAL VERSION PERFORMANCE

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

Voting and instrumentation led to the detection of the same number of faults in total.

There are some inherent problems in comparing these two techniques. Self checks are capable of detecting errors caused by faults that may not actually result in a failure for that particular execution. In the area of testing, this is extremely valuable since the goal is to find all faults that could possibly cause failures and remove them. It could be argued, however, that for run-time fault tolerance, error detection in nonfailing programs is of lesser importance and could actually be a hindrance. The above 8 faults detected by self checks and not by voting include two faults that were not detected by voting because they did not cause failures during the execution of the programs. However, these two faults did cause failures and were detected during the previous execution of one million inputs for these same programs. The self checks found errors caused by one previously-unknown fault that did cause a failure in the 100 000 input cases that was not detected by voting. The other five faults detected by self checks alone did not cause failures for the 100 000 input cases or for the previous one million input cases.

Another problem comparing the techniques involves the differences in effectiveness or "coverage" of the fault detection. The effectiveness or coverage of the technique in detecting faults is defined as how often an error was detected given that it occurred and a potentially effective check was made. Table IX credits a technique with detecting an error even if it does not detect the error every time it occurs.

Tables X(a) and X(b) show the fault-detection coverage for each of the voted triples and pairs. 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 self checks 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 and the pair detected these failures. Two faults (i.e., 20.2 and 25.1) that caused failures on the original one-million input cases did not cause failures for these 100 000 input cases (although the self checks 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 and the pair. 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 pair detected that particular fault given that it caused an erroneous result. For triples, this ranged from 29 to 100%, with an average of 0.72 and a standard deviation of 0.29. For 2-version voting, the results are a little better (as would be expected) but the coverage still averaged only 0.84 (standard deviation of 0.18) with a couple of pairs as low as 0.57.

TABLE IX
COMPARISON OF FAULT DETECTION

Faults	Voting	Instrumentation
3.1	√	
3.2	√	
3.3	√	√
3.4	√	
6.1	√	√
6.2	√	√
6.3	√	
6.4		√
8.1	√	√
8.2	√	√
8.3		√
12.1	√	√
12.2	√	
12.3		√
12.4		√
14.1	√	
14.2	√	
20.1	√	
20.2		√
20.3		√
23.1	√	√
23.2	√	√
25.1		√
25.2	√	√
25.3	√	√
25.4		√
total	18	18

The partial detection of faults by voting 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 100 000 input cases executed in this part of the experiment and for the previous one million input cases except for version 23a where, as discussed earlier, the participant used *N*-version voting to implement the self checking. 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

TABLE X
(a) FAULT-DETECTION COVERAGE AMONG THE VOTING TRIPLES. (b) FAULT-DETECTION COVERAGE AMONG THE VOTING PAIRS.

3-Way Voting

faults	3.1	3.2	3.3	3.4	6.1	6.2	6.3	8.1	8.2	12.1	12.2	14.1	14.2	20.1	20.2	23.1	23.2	25.1	25.2	25.3
100K Cases	10	15	61	137	5	50	6	20	5	42	7	133	7	25	0	1	3	0	8	2
3/6/8	1*	0*	61	137	5	50	6	1*	0*	.	.	.	.	.	.	.	.	.	.	.
3/6/12	10	15	61	137	5	50	6	.	.	42	7	.	.	.	.	.	.	.	.	.
3/6/14	10	13*	61	137	5	50	6	.	.	.	.	131*	7	.	.	.	.	.	.	.
3/6/20	1*	0*	61	137	5	50	6	.	.	.	.	.	.	1*	.	.	.	.	.	.
3/6/23	10	14*	61	137	5	50	6	.	.	.	.	.	.	.	.	0*	3	.	.	.
3/6/25	6*	12*	61	137	5	50	6	.	.	.	.	.	.	.	.	.	.	.	1*	2
3/8/12	1*	0*	61	137	.	.	.	1*	0*	42	7	.	.	.	.	.	.	.	.	.
3/8/14	1*	0*	61	137	.	.	.	1*	0*	.	.	131*	7	.	.	.	.	.	.	.
3/8/20	0*	0*	61	137	.	.	.	0*	0*	.	.	.	.	0*	.	.	.	.	.	.
3/8/23	0*	0*	61	137	.	.	.	0*	0*	.	.	.	.	.	.	0*	2*	.	.	.
3/8/25	1*	0*	61	137	.	.	.	1*	0*	.	.	.	.	.	.	.	.	.	0*	2
3/12/14	10	13*	61	137	.	.	.	.	.	42	0*	131*	0*	.	.	.	.	.	.	.
3/12/20	1*	0*	61	137	.	.	.	.	.	42	7	.	.	1*	.	.	.	.	.	.
3/12/23	10	14*	61	137	.	.	.	.	.	42	7	.	.	.	.	0*	3	.	.	.
3/12/25	6*	12*	61	137	.	.	.	.	.	42	7	.	.	.	.	.	.	.	1*	2
3/14/20	1*	0*	61	137	.	.	.	.	.	.	.	131*	7	1*	.	.	.	.	.	.
3/14/23	10	12*	61	137	.	.	.	.	.	.	.	131*	7	.	.	0*	3	.	.	.
3/14/25	6*	10*	61	137	.	.	.	.	.	.	.	131*	7	.	.	.	.	.	1*	2
3/20/23	0*	0*	61	137	.	.	.	.	.	.	.	.	.	0*	.	0*	2*	.	.	.
3/20/25	1*	0*	61	137	.	.	.	.	.	.	.	.	.	1*	.	.	.	.	0*	2
3/23/25	6*	11*	61	137	.	.	.	.	.	.	.	.	.	.	.	0*	3	.	1*	2
6/8/12	.	.	.	.	5	50	6	20	5	42	7	.	.	.	.	.	.	.	.	.
6/8/14	.	.	.	.	5	50	6	20	3*	.	.	131*	7	.	.	.	.	.	.	.
6/8/20	.	.	.	.	5	50	6	0*	0*	.	.	.	.	0*	.	.	.	.	.	.
6/8/23	.	.	.	.	5	50	6	18*	5	.	.	.	.	.	.	0*	2*	.	.	.
6/8/25	.	.	.	.	5	50	6	14*	4*	.	.	.	.	.	.	.	.	.	1*	2
6/12/14	.	.	.	.	5	50	6	.	.	42	0*	133	0*	.	.	.	.	.	.	.
6/12/20	.	.	.	.	5	50	6	.	.	42	7	.	.	25	.	.	.	.	.	.
6/12/23	.	.	.	.	5	50	6	.	.	42	7	.	.	.	.	1	3	.	.	.
6/12/25	.	.	.	.	5	50	6	.	.	42	7	.	.	.	.	.	.	.	8	2
6/14/20	.	.	.	.	5	50	6	.	.	.	.	131*	7	23*	.	.	.	.	.	.
6/14/23	.	.	.	.	5	50	6	.	.	.	.	133	7	.	.	1	3	.	.	.
6/14/25	.	.	.	.	5	50	6	.	.	.	.	133	7	.	.	.	.	.	8	2
6/20/23	.	.	.	.	5	50	6	.	.	.	.	.	.	23*	.	0*	2*	.	.	.
6/20/25	.	.	.	.	5	50	6	.	.	.	.	.	.	18*	.	.	.	.	1*	2
6/23/25	.	.	.	.	5	50	6	.	.	.	.	.	.	.	.	1	3	.	8	2
8/12/14	.	.	.	.	.	.	.	20	3*	42	0*	131*	0*	.	.	.	.	.	.	.
8/12/20	.	.	.	.	.	.	.	0*	0*	42	7	.	.	0*	.	.	.	.	.	.
8/12/23	.	.	.	.	.	.	.	18*	5	42	7	.	.	.	.	0*	2*	.	.	.
8/12/25	.	.	.	.	.	.	.	14*	4*	42	7	.	.	.	.	.	.	.	1*	2
8/14/20	.	.	.	.	.	.	.	0*	0*	.	.	131*	7	0*	.	.	.	.	.	.
8/14/23	.	.	.	.	.	.	.	18*	3*	.	.	131*	7	.	.	0*	2*	.	.	.
8/14/25	.	.	.	.	.	.	.	14*	2*	.	.	131*	7	.	.	.	.	.	1*	2
8/20/23	.	.	.	.	.	.	.	0*	0*	.	.	.	.	0*	.	0*	2*	.	.	.
8/20/25	.	.	.	.	.	.	.	0*	0*	.	.	.	.	0*	.	.	.	.	0*	2
8/23/25	.	.	.	.	.	.	.	12*	4*	.	.	.	.	.	.	0*	2*	.	1*	2
12/14/20	.	.	.	.	.	.	.	.	.	42	0*	131*	0*	23*	.	.	.	.	.	.
12/14/23	.	.	.	.	.	.	.	.	.	42	0*	133	0*	.	.	1	3	.	.	.
12/14/25	.	.	.	.	.	.	.	.	.	42	0*	133	0*	.	.	.	.	.	8	2
12/20/23	.	.	.	.	.	.	.	.	.	42	7	.	.	23*	.	0*	2*	.	.	.
12/20/25	.	.	.	.	.	.	.	.	.	42	7	.	.	18*	.	.	.	.	1*	2
12/23/25	.	.	.	.	.	.	.	.	.	42	7	.	.	.	.	1	3	.	8	2
14/20/23	.	.	.	.	.	.	.	.	.	.	.	131*	7	21*	.	0*	2*	.	.	.
14/20/25	.	.	.	.	.	.	.	.	.	.	.	131*	7	16*	.	.	.	.	1*	2
14/23/25	.	.	.	.	.	.	.	.	.	.	.	133	7	.	.	1	3	.	8	2
20/23/25	.	.	.	.	.	.	.	.	.	.	.	.	.	16*	.	0*	2*	.	1*	2
	44%	40%	100%	100%	100%	100%	100%	41%	36%	100%	71%	99%	71%	40%		29%	83%		36%	100% (mean : 71.7%, sd 29.3)

(a)

2-Way Voting

faults	3.1	3.2	3.3	3.4	6.1	6.2	6.3	8.1	8.2	12.1	12.2	14.1	14.2	20.1	20.2	23.1	23.2	25.1	25.2	25.3
100K Cases	10	15	61	137	5	50	6	20	5	42	7	133	7	25	0	1	3	0	8	2
3/6	10	15	61	137	5	50	6	.	.	.	.	.	.	.	.	.	.	.	.	.
3/8	1*	0*	61	137	.	.	.	1*	0*	.	.	.	.	.	.	.	.	.	.	.
3/12	10	15	61	137	.	.	.	.	.	42	7	.	.	.	.	.	.	.	.	.
3/14	10	13*	61	137	.	.	.	.	.	.	.	131*	7	.	.	.	.	.	.	.
3/23	9*	15*	61	137	.	.	.	.	.	.	.	.	.	.	.	0*	3	.	.	.
3/20	1*	0*	61	137	.	.	.	.	.	.	.	.	.	1*	.	.	.	.	.	.
3/25	6*	12*	61	137	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1*	2
6/8	.	.	.	.	5	50	6	20	5	.	.	.	.	.	.	.	.	.	.	.
6/12	.	.	.	.	5	50	6	.	.	42	7	.	.	.	.	.	.	.	.	.
6/14	.	.	.	.	5	50	6	.	.	.	.	133	7	.	.	.	.	.	.	.
6/20	.	.	.	.	5	50	6	.	.	.	.	.	.	25	.	.	.	.	.	.
6/23	.	.	.	.	5	50	6	.	.	.	.	.	.	.	.	1	3	.	.	.
6/25	.	.	.	.	5	50	6	.	.	.	.	.	.	.	.	.	.	.	8	2
8/12	.	.	.	.	.	.	.	20	5	42	7	.	.	.	.	.	.	.	.	.
8/14	.	.	.	.	.	.	.	20	3*	.	.	131*	7	.	.	.	.	.	.	.
8/20	.	.	.	.	.	.	.	0*	0*	.	.	.	.	0*	.	.	.	.	.	.
8/23	.	.	.	.	.	.	.	18*	5	.	.	.	.	.	.	0*	2*	.	.	.
8/25	.	.	.	.	.	.	.	14*	4*	.	.	.	.	.	.	.	.	.	1*	2
12/14	.	.	.	.	.	.	.	.	.	42	0*	133	0*	.	.	.	.	.	.	.
12/20	.	.	.	.	.	.	.	.	.	42	7	.	.	25	.	.	.	.	.	.
12/23	.	.	.	.	.	.	.	.	.	42	7	.	.	.	.	1	3	.	.	.
12/25	.	.	.	.	.	.	.	.	.	42	7	.	.	.	.	.	.	.	8	2
14/20	.	.	.	.	.	.	.	.	.	.	.	131*	7	23*	.	.	.	.	.	.
14/23	.	.	.	.	.	.	.	.	.	.	.	133	7	.	.	1	3	.	.	.
14/25	.	.	.	.	.	.	.	.	.	.	.	133	7	.	.	.	.	.	8	2
20/23	.	.	.	.	.	.	.	.	.	.	.	.	.	23*	.	0*	2*	.	.	.
20/25	.	.	.	.	.	.	.	.	.	.	.	.	.	18*	.	.	.	.	1*	2
23/25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	3	.	8	2
	67%	67%	100%	100%	100%	100%	100%	66%	63%	100%	86%	99%	86%	66%		57%	90%		57%	100% (mean: 83.6%, sd: 17.5)

(b)

and other parts of the internal state. As a result, simultaneous failures of multiple versions sometimes caused voting to fail to detect erroneous results.

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 and thus decreasing the amount of error detection and fault tolerance possible. At the extreme, this results in totally specified and thus identical versions.

Simultaneous failure does not explain all the faults missed by voting. Notice 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 the errors caused by these faults (and six other faults that voting did not detect even on the million input cases) again 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 errors caused by faults that back-to-back testing (i.e., using the comparison of the results of multiple versions as a test oracle) [6], [7], [21], [23] 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 [26].

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 this team approach would be profitable.

IV. Conclusions

Almost no empirical data exists on the effectiveness of using self checks to detect errors, and no previous studies have compared error and fault detection using *N*-version voting with self checks. Several results were obtained from this study that should guide us and others in the evaluation of current proposals for error detection and fault tolerance, in the design of new techniques, 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 are very subtle. Out of a possible 60 known faults (i.e., 20 known faults in the 8 versions with each version given to three people) 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 by the three programmers working on the same initial version, 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 postconditions 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 the errors caused by the 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 errors caused by 6 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 may be less effective in finding faults than verifying the validity of the intermediate results and structures of the program.

Specification-based checks alone were not as effective as using them together with code-based checks. This suggests 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 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 errors caused by a particular fault compared to the 21 voting triples and 7 voting pairs that could

possibly find each fault, self checking led to the detection of 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 fact that each found faults that were not detected by the other suggests 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. This result has been replicated by two of the authors in a subsequent experiment using different programs and a variety of testing techniques [27].

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 errors 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 [19] suggest that safety analysis using software fault trees [18] 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.

Acknowledgment

The authors are pleased to acknowledge the efforts of the experiment participants: D. W. Aha, T. Bair, J. Beusmans, B. Catron, H. S. Delugach, S. Emadi, L. Fitch, W. A. Frye, J. Gresh, R. Jones, J. R. Kipps, F. Leifman, C. Livadas, J. Marco, D. A. Montuori, J. Palesis, N. Pomicter, M. T. Roberson, K. Ruhleder, B. Gates Spielman, Y. Venkata Srinivas, T. Strayer, G. R. Taylor III, and R. 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," in *Dig. Papers FTCS-15: Fifteenth Annu. Symp. Fault-Tolerant Computing*, Ann Arbor, MI, June 1985, pp. 140–145.

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

[3] D. M. Andrews, "Using executable assertions for testing and fault tolerance," in *Proc. Ninth Int. Symp. Fault-Tolerant Computer Systems*, June 1979, pp. 102–105.

[4] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, Aug. 1984.

[5] J. P. Benson and S. H. Saib, "A software quality assurance experiment," in *Proc. Software Quality and Assurance Workshop*, Nov. 1978, pp. 87–91.

[6] P. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti, and S. Yoshimura, "PODS—A project on diverse software," *IEEE Trans. Software Eng.*, vol. SE-12, no. 9, pp. 929–940, Sept. 1986.

[7] S. S. Brilliant, "Testing software using multiple versions," Ph.D. dissertation, Univ. Virginia, Sept. 1987.

[8] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an *N*-version software experiment," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 238–247, Feb. 1990.

[9] ——, "The consistent comparison problem in *N*-version software," *IEEE Trans. Software Eng.*, vol. 15, no. 11, pp. 1481–1485, Nov. 1989.

[10] L. Chen and A. Avizienis, "*N*-version programming: A fault-tolerance approach to reliability of software operation," in *Dig. Papers FTCS-8: Eighth Annu. Symp. Fault Tolerant Computing*, Toulouse, France, June 1978, pp. 3–9.

[11] J. R. Dunham, "Software errors in experimental systems having ultra-reliability requirements," in *Dig. Papers FTCS-16: Sixteenth Annu. Symp. Fault-Tolerant Computing*, Vienna, Austria, July 1986, pp. 158–164.

[12] L. Gmeiner and U. Voges, "Software diversity in reactor protection system: An experiment," in *Proc. IFAC Workshop SAFECOMP '79*, 1979, pp. 75–79.

[13] D. Gries, *The Science Of Programming*. New York: Springer-Verlag, 1981.

[14] A. L. Hopkins, *et al.*, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proc. IEEE*, vol. 66, pp. 1221–1239, Oct. 1978.

[15] J. P. J. Kelly *et al.*, "Multi-version software development," in *Proc. IFAC Workshop Safecomp '86*, Sarlat, France, Oct. 1986, pp. 43–49.

[16] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 96–109, Jan. 1986.

[17] ——, "An empirical study of failure probabilities in multi-version software," in *Dig. Papers FTCS-16: Sixteenth Annu. Symp. Fault-Tolerant Computing*, Vienna, Austria, July 1986, pp. 165–170.

[18] N. G. Leveson and P. R. Harvey, "Analyzing software safety," *IEEE Trans. Software Eng.*, vol. SE-9, no. 5, pp. 569-579, Sept. 1983.
[19] N. G. Leveson, and T. J. Shimeall, "Safety assertions for process-control systems," in *Dig. Papers FTCS-13: Thirteenth Annu. Symp. Fault-Tolerant Computing*, Milan, Italy, June 1983, pp. 236-240.
[20] H. Partsch and R. Steinbrüggen, "Program transformation systems," *ACM Comput. Surveys*, vol. 15, no. 3, pp. 199-236, Sept. 1983.
[21] C. V. Ramamoorthy, Y. K. Mok, E. B. Bastani, G. H. Chin, and K. Suzuki, "Application of a methodology for the development and validation of reliable process control software," *IEEE Trans. Software Eng.*, vol. SE-7, no. 6, pp. 537-555, Nov. 1981.
[22] B. Randell, "System structure for software fault-tolerance," *IEEE Trans. Software Eng.*, 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," in *Proc. Safecomp '86*, Sarlat, France, Oct. 1986, pp. 27-34.
[24] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, pp. 222-238, Aug. 1983.
[25] K. R. Scott, J. W. Gault, D. F. McAllister, and J. Wiggs, "Experimental validation of six fault tolerant software reliability models," in *Dig. Papers FTCS-14: Fourteenth Annu. Symp. Fault-Tolerant Computing*, Kissemmee, NY, 1984, pp. 102-107.
[26] T. J. Shimeall and N. G. Leveson, "An empirical comparison of software fault tolerance and fault elimination," in *Proc. 2nd Workshop Software Testing, Verification, and Analysis*, Banff, Alta., Canada, July 1988, pp. 180-187.
[27] ——, "An empirical comparison of software fault tolerance and fault elimination," Naval Postgraduate School, Monterey, CA, Tech. Rep. NPS52-89-D47, July 1989.
[28] L. G. Stucki, "New directions in automated tools for improving software quality," in *Current Trends in Programming Methodology—volume II: Program Validation*. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 80-111.
[29] J. H. Wensley *et al.*, "SIFT, the design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 66, pp. 1240-1254, Oct. 1978.

Nancy G. Leveson received the B.A. degree in mathematics, the M.S. degree in management, and the Ph.D. degree in computer science from the University of California, Los Angeles.

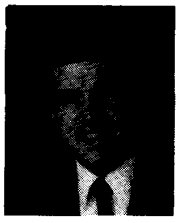
She has worked for IBM and is currently an Associate Professor of Computer Science at the University of California, Irvine. Her current interests are in software reliability, software safety, and software fault tolerance. She heads the Software Safety Project at UCI which is exploring a range of software engineering topics involved in specifying, designing, verifying, and assessing reliable and safe real-time software.

Dr. Leveson is a member of the Association for Computing Machinery, the IEEE Computer Society, and the System Safety Society.

Stephen S. Cha received the B.S. and M.S. degrees in information and computer science from the University of California, Irvine, in 1983 and 1986, respectively.

He is currently a Ph.D. candidate at UCI. His research interests include software safety and software fault-tolerance.

Mr. Cha is a student member of the IEEE Computer Society.



John C. Knight received the B.Sc. degree in mathematics from the Imperial College of Science and Technology, London, England, and the Ph.D. degree in computer science from the University of Newcastle-upon-Tyne, Newcastle-upon-Tyne, England, in 1969 and 1973, respectively.

From 1974 to 1981 he was with NASA's Langley Research Center and he joined the Department of Computer Science at the University of Virginia, Charlottesville, in 1981. He spent the period from August 1987 to August 1989 on leave at the Software Productivity Consortium in Herndon, VA.

Dr. Knight is a member of the Association for Computing Machinery and the IEEE Computer Society.

Timothy J. Shimeall received the Ph.D. degree in information and computer science from the University of California, Irvine, in March 1989.

Since September 1988, he has been an Assistant Professor at the Naval Postgraduate School in Monterey, CA. His research interests are software testing and software safety.