SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2002; **12**:155–171 (DOI: 10.1002/stvr.248)

An empirical evaluation of six methods to detect faults in software

Sun Sup So[1], Sung Deok Cha[2], Timothy J. Shimeall[3,*,†] and Yong Rae Kwon[4]

[1]*Department of Computer Information Engineering, Division of Computer and Media Information Engineering, Kongju National University, Kongju, Korea*
[2]*Department of Electrical Engineering & Computer Science (EECS) and Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology, Taejon, Korea*
[3]*CERT® Analysis Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, U.S.A.*
[4]*Department of Electrical Engineering & Computer Science (EECS), Korea Advanced Institute of Science and Technology, Taejon, Korea*

SUMMARY

Although numerous empirical studies have been conducted to measure the fault detection capability of software analysis methods, few studies have been conducted using programs of similar size and characteristics. Therefore, it is difficult to derive meaningful conclusions on the relative detection ability and cost-effectiveness of various fault detection methods. In order to compare fault detection capability objectively, experiments must be conducted using the same set of programs to evaluate all methods and must involve participants who possess comparable levels of technical expertise. One such experiment was 'Conflict1', which compared voting, a testing method, self-checks, code reading by stepwise refinement and data-flow analysis methods on eight versions of a battle simulation program. Since an inspection method was not included in the comparison, the authors conducted a follow-up experiment 'Conflict2', in which five of the eight versions from Conflict1 were subjected to Fagan inspection. Conflict2 examined not only the number and types of faults detected by each method, but also the cost-effectiveness of each method, by comparing the average amount of effort expended in detecting faults. The primary findings of the Conflict2 experiment are the following. First, voting detected the largest number of faults, followed by the testing method, Fagan inspection, self-checks, code reading and data-flow analysis. Second, the voting, testing and inspection methods were largely complementary to each other in the types of faults detected. Third, inspection was far more cost-effective than the testing method studied. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: experiments; software testing; code reading; self-checks; Fagan inspection; N-version voting

*Correspondence to: Timothy J. Shimeall, CERT® Analysis Center, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, U.S.A.
†E-mail: tjs@cert.org

Contract/grant sponsor: Korea Science and Engineering Foundation (KOSEF)
Contract/grant sponsor: Naval Postgraduate School Research Council
Contract/grant sponsor: Department of Defense

Received 1 May 1998
Accepted 25 October 2001

Published online 22 May 2002

1. INTRODUCTION

As software unreliability has become a major bottleneck in improving system reliability [1], developers find it essential to detect and eliminate software faults at the earliest possible opportunity. Many fault detection methods have been proposed in the literature. The software industry uses methods such as testing [2], voting (also known as back-to-back testing) [3], inspections [4,5], walk-throughs [6], self-checking code (also known as assertions) [7,8], and data-flow analysis [9]. While one might suggest that some or all of the above techniques are to be used to develop the highest quality software possible, software development projects will always have to be carried out with limited resources. In order to maximize software development productivity, practitioners need to be guided by empirical and objective data on the relative cost-effectiveness of proposed methods. Unfortunately, most of the empirical studies to date have been conducted under different conditions and have examined programs of different sizes and application domains.

To avoid bias in the comparison of methods, the same set of programs should be used in the comparison and participants should possess comparable levels of technical skills. Only a small number of controlled experiments satisfying these criteria have been conducted, including the three LIP (Launch Interceptor Program experiments; in this paper, henceforth referred to as 'LIP1', 'LIP2', and 'LIP3') [10–12] and the Conflict [13,14] (in this paper, henceforth referred to as 'Conflict1') experiments. In the Conflict1 experiment, Shimeall used eight versions of a battle simulation program (in this paper, henceforth referred to as the 'Conflict versions') to compare fault detection abilities of voting, self-checking, requirements-based and design-based testing, code reading by stepwise refinement, and static data-flow analysis. This paper describes results obtained from a follow-up study to the Conflict1 experiment in which five of the eight Conflict versions were subjected to Fagan inspection.

2. RELATED WORK

This section describes four experiments that form a backdrop for the Conflict2 experiment. Three of these experiments, LIP1, LIP2, and LIP3, used a set of programs that mimicked the decision logic regarding the launch of an interceptor missile, and are referred to as the LIP experiments. The fourth experiment, Conflict1, used a set of programs that implemented a simplified combat simulation program.

2.1. The LIP experiments

The LIP were originally developed in the LIP1 experiment to examine empirically the truth of the independent failure assumption of N-version software. N-version programming and recovery blocks, based on the concept of redundancy, are the two most widely known techniques for software fault tolerance. These methods attempt to develop software that continues to operate correctly at run-time, despite failures encountered by some of the redundant versions. The LIP1 experiment [12] used 27 LIP versions, developed by seniors and graduate students majoring in computer science. The development started with a common specification written in English. The LIP versions were then executed with one million input cases, randomly generated using a realistic operational profile.

The results were then combined to simulate two- and three-version voting on all possible combinations. The reliability improvement figures were significantly lower than expected from the figure derived based on the independent failure assumption. Therefore, the experiment administrators concluded that, with a probability of coincidence at less than 1%, the hypothesis of independent version failure should be rejected.

Fault detection is the common and initial step, both in providing fault tolerance in software and in eliminating faults from software. In N-version programming, voting is used as the fault detection mechanism; acceptance tests are the corresponding mechanism for the recovery block. Since acceptance tests are similar in concept and structure to self-checks, Leveson *et al.* [10] conducted the LIP2 experiment to compare the fault detection ability of voting and self-checks. In this second study, eight of the 27 LIP versions were randomly selected from the versions that were known to contain faults. There were 24 graduate students majoring in computer science hired to insert self-checks in the LIP versions, based on separate readings of the specification and the code. The participants had an average of 2.3 years of graduate study and 1.7 years of industrial experience. Each LIP version was assigned to three participants. There were no restrictions on the number or type of self-checks each participant could add. The results indicated that voting and self-check methods were largely comparable in that each method detected the same total number of faults—11. Moreover, each method detected an equal number of faults—seven—that the other method failed to detect.

So *et al.* [11] conducted the LIP3 study to compare the effectiveness of an inspection method to that of voting and self-checks. The Fagan inspection method was selected as a representative sample of the various inspection methods. This experiment was carried out at the Korea Advanced Institute of Science and Technology (KAIST) as part of a graduate-level course on software engineering. The participants had an average of 1.8 years of graduate study as well as 2.5 years of industrial experience. The participants in the LIP1, LIP2, and LIP3 experiments all had a similar level of experience. There were several resource constraints inherent in the LIP3 experimental design. For example, the version authors were absent from all inspection teams due to the geographic and time separations between the original experiment and the LIP3 experiment. Furthermore, of the eight LIP versions used in the LIP2 experiment, only four were subjected to Fagan inspection in the LIP3 experiment, due to a limited number of students enrolled in the course. Data from the LIP3 study indicated that inspection is a more effective method than voting or self-checks; the inspection method detected 14 faults, whereas both voting and self-checks detected only 11 faults. The administrators of the LIP3 study also used its time-sheet data to compare the cost-effectiveness of self-checks and inspection. Voting was excluded in this comparison, since the voting process is largely mechanical, although there is clearly a cost associated with the process. Data from LIP3[‡] revealed that the inspection method required an average of 3.14 staff hours to detect a fault, whereas the self-checking method took an average of 40.3 staff hours.

Although the LIP3 experiment [11] was the first controlled experiment in which voting, self-checks, and inspection methods were compared for the same set of versions, there were several limitations that motivated planning of another experiment. First, the versions used in the three LIP experiments consisted of only 500 to 900 lines of code, including comments, written in Pascal. A desire for realism

[‡]Readers are encouraged to consult the LIP3 paper [11] for more explicit discussion of these results; the tabular data later in this paper derive from the Conflict1 and Conflict2 experiments and do not pertain to the LIP experiments.

would warrant extending the experiment using larger programs. Second, voting and self-checks are not the fault detection techniques most widely used in industry. Providing empirical results useful to practitioners would require a comparison of the effectiveness of testing techniques and inspection techniques. Third, in the LIP3 experiment, each inspection team performed a Fagan inspection on all four versions of LIP software chosen for the experiment. Therefore, it is likely that the inspection teams gained experience while the experiment was in progress and may have detected more faults when inspecting later versions. The experiment administrators wanted to extend the study in a way that would avoid potential bias in the later results due to learning effects.

2.2. The Conflict1 experiment

2.2.1. Development

In the Conflict1 experiment [13] five fault detection methods were applied to a set of eight versions of a combat simulation program, written in Pascal. The common specifications for this program were written in English, supplemented by a mathematical notation, describing a simplified land combat environment, with two forces located in three-dimensional terrain, moving, observing, contending, and recovering. Iteration among these actions was specified across forces and across simulated time. The output of the program was a final status of the forces and a cumulative record of their interactions. The specification required the use of a specific set of data structures for input and output, with a supplied shell program to load and store these data structures. The versions were all designed using structured programming methodologies. The developers were all senior computer science undergraduates who had been introduced to these design practices in several courses during their studies. To maximize potential design diversity, the experiment administrator deliberately offered several alternative design methods and allowed the development participants to select among them. The resulting designs followed a functional decomposition of the specification, and were quite iterative. All of the versions show a decomposition reflecting the location, movement, observation, attrition, and recovery functions described in the specification, although the arrangement of this decomposition and the degree of further decomposition varies from version to version. The amount of coupling between modules in the versions varies greatly, as does the degree of cohesion within version modules. The looping structure of each version varies from version to version. The resulting versions vary in length from 1500 to 7500 lines (including comments).

The Conflict versions are numbered in the order of their submission to the experiment administrator for acceptance into the Conflict1 experiment. There is no correlation between length and version number, nor, due to varying work schedules, between time spent in development and version number. Indeed, the numbering appears to be arbitrary with respect to measurable characteristics of the version. A 15-data set acceptance test, developed prior to design and implementation, was used to ensure a very minimal level of functionality in each version. Eight versions passed this acceptance test.

2.2.2. Self-checks

While designing and coding, the programmers instrumented their software with self-checks based on training in the construction and placement of self-checks using sections from a fault tolerance textbook [15]. Self-checks, which are executable conditional statements, act as tests on the internal

state of the software. The self-checks examine that state for anticipated erroneous conditions. In the case of the Conflict1 experiment, detection of an erroneous condition led to the generation of a message. No requirement was made of the programmers as to which forms of self-checks should be inserted, nor as to where they should be inserted. The programmers were told that at least one self-check was required, and not to include any recovery code for dealing with errors detected by the self-checks.

2.2.3. Voting

Once the versions had been developed and had successfully passed a 15-data set acceptance test, the versions were all executed and voted on 10 000 randomly generated data sets. The profile for the random data generation was based on expert guidance as to realistic data ranges, derived from a TRW combat simulation program. In this experiment, there were eight Conflict versions that were voted, considered both two at a time and three at a time. To facilitate identification of faults detected by the voting, a 'tinsel version' (a not-completely-trusted source of comparison data) was created by the experiment administrator and the results of the tinsel version were compared with the student version results. Any disagreement with the tinsel version was evaluated, as were all assertion messages, and any detected faults were identified via debugging.

2.2.4. Static analysis

In parallel with this activity, the Conflict versions were evaluated using static data-reference anomaly analysis [9]. This analysis examines the source code of the software to identify uninitialized data, data initialized but not later used, and unused variables, on a path-by-path basis. This analysis is entirely automated. The automated tool used in the Conflict1 experiment was a specifically-constructed implementation of the Fosdick and Osterweil algorithms [9], applied to the Pascal used in the experiment. Each anomaly identified using these algorithms was evaluated and any faults validated via desk-checking.

2.2.5. Code reading

While the voting and static analysis were underway, a group of students distinct from the programmers and unaware of the structure of any of the versions was trained in code reading by stepwise abstraction [16]. None of the readers had practiced any systematic code reading prior to this experiment. The readers had levels of experience comparable to the programmers. The readers were each assigned one version and provided hard copy of only that version. They were required to annotate the version with abstractions of the purpose of each code section and to identify any faults detected. No time limits were placed on the readers, but they were required to keep a log of their hours. When the readers had finished, each of the faults they had detected was evaluated by desk-checking or debugging.

2.2.6. Testing

Finally, a third group of students (testers), disjoint from the programmers and readers but with equivalent levels of experience, was trained in software testing using both requirements-based [6] and design-based testing [17] methods. The students built test cases to evaluate each requirement at least

Table I. Number of faults[a] detected (parentheses indicate the number of faults detected by no other method).

	Versions								Method total
	1	2	3	4	5	6	7	8	
Voting	14(11)	11(9)	19(12)	11(6)	21(14)	15(8)	15(6)	17(12)	123(78)
Testing	9(1)	15(10)	19(16)	18(11)	22(12)	10(0)	17(11)	11(10)	121(71)
Self-checking	11(4)	4(3)	5(1)	12(8)	11(2)	6(2)	6(2)	5(4)	60(26)
Code reading	2(0)	5(2)	6(4)	3(2)	1(0)	1(1)	1(0)	19(15)	38(24)
Data-flow analysis	0	0	2(1)	0	0	1(1)	2(0)	0	5(2)
Version total	36	35	51	44	55	33	41	52	347

[a]In the course of doing the follow-up experiment, several errors were identified in the results of the Conflict1 experiment as reported [13]. These errors have been corrected in the table shown here.

once (100% requirements coverage), based on a detailed re-evaluation of the requirements that isolated each separately-observable characteristic described in the specification, and then partitioned them into testable and non-testable requirements. Each testable requirement was included in the coverage goal. The testers were then asked to bring the level of design coverage up to the all-p-uses level, at 100% coverage, as measured by a software tool constructed by Frankl and Weyuker [17]. (For convenience, this process involving both requirements-based and design-based components is henceforth referred to as 'the testing method'. Separate results for each component are not available.) A total of 97 test cases were constructed by the testers. All test cases were executed on all Conflict versions, and the results evaluated by comparison with the 'tinsel version'. Any disagreements between a Conflict version and the tinsel version were evaluated and the detected faults identified via debugging.

2.2.7. Findings of Conflict1

A primary finding of the Conflict1 experiment, shown in Table I, was that the voting method and the testing method were shown to be the two most effective methods. Only faults relating to the correctness of results were considered in the Conflict1 experiment: maintainability was ignored by all participants and efficiency was considered only if a specific code section caused the execution of a data set to exceed the normal 15-minute (or less) run-time by more than one week[§]. Voting and the testing method detected 123 and 121 total faults scattered in eight Conflict versions, respectively. However, all methods in the Conflict1 experiment were largely complementary in that more than half of the faults detected by each method were not detected by any of the other methods. The numbers in parentheses, referring to faults detected only by the specified method, show this complementary result in Table I. For example, voting detected 11 faults in Version 2, and nine of those faults were not detected

[§]This arbitrary choice resulted in only two sections of code being considered faults in the Conflict1 experiment; both cases involved a division being implemented via iterated subtraction.

by any other method. Self-checking and code reading detected 60 and 38 total faults, respectively—considerably less effective than voting or the testing method. The static data-flow analysis method proved to be the least effective, detecting only five total faults. The 'Version total' row is simply the sum of the faults detected by each method, not the total number of known faults in the version at the end of the Conflict1 experiment (i.e. the data in this row do not eliminate duplicate detections). See Table II, presented later, in Section 3.3 of this paper, for the total number of known faults (the size of the union of fault detections) in the versions used in the Conflict2 experiment.

3. THE Conflict2 EXPERIMENT

This section describes the follow-up experiment, referred to here as 'Conflict2', in which five of the eight versions from Conflict1 were subject to Fagan inspections.

3.1. Fagan inspections

The Fagan inspection process involves systematic group review of code or related artifacts such as requirements and design documents. An inspection team is most productive when its team members work in harmony and fulfil their assigned roles. Inspection has been proven successful on several large-scale industrial applications and is in widespread use in industry [18].

A typical Fagan inspection team consists of a moderator, reader, inspector and author. The moderator plays an especially important leadership role and must ensure that the team stays focused on detecting faults without being sidetracked (e.g., suggesting necessary corrections or desirable enhancements). The reader's role is to paraphrase the product being reviewed at a reasonable pace. Otherwise, an inspection team might be tempted to inspect software too quickly and superficially. The author's presence in the inspection meetings is generally considered beneficial because (1) the author can assist the inspection team to understand the product better, and (2) the author is more prepared to understand the exact nature of the faults the inspection team finds. An inspector's role is to examine the software from a tester's viewpoint.

The Fagan inspection process consists of the following steps, each with specific objectives: planning, overview, preparation, inspection, rework, and follow-up.

- **Planning:** When materials to be inspected pass their entry criteria (i.e. source code successfully compiles without syntax errors), inspection team members are selected, and inspection schedules (e.g., time and place) are established.
- **Overview:** Team members are briefed on the material to be inspected, and roles are assigned.
- **Preparation:** Team members review the material individually to prepare themselves to fulfil the assigned roles.
- **Inspection:** The team conducts an inspection meeting to find faults, and records the faults detected. The purpose of the inspection meeting is detection of faults or style violations, and any attempts to find alternative solutions should be strongly discouraged by the moderator.
- **Rework:** The author reviews the record of faults detected, clarifying which are actually faults and which are misunderstandings in the inspection process. The author then modifies the code to correct the faults.

- **Follow-up:** The moderator or the entire inspection team reviews the product again to ensure that all fixes are effective and that no additional defects have been introduced during rework.

Several variations have been proposed to the Fagan inspection method [19–21]. However, the authors chose to apply the inspection method as described by Fagan [5], since it is the most widely used inspection method in industry. The experiment administrators and participants fully adhered to the principles and recommendations of the Fagan inspection process as closely as possible.

3.2. Experimental design of Conflict2

The Conflict inspection experiment, 'Conflict2', was carried out as a term project in a graduate-level software engineering course at KAIST. The 15 participants in this experiment had no previous experience in performing Fagan inspections or any knowledge of the Conflict versions or their faults. All but two participants were graduate students majoring in computer science who had participated in small- to medium-sized software development projects as a part of their undergraduate curriculum (i.e. projects with a maximum of several thousand lines of code). Due to the limited number of participants, only three to five inspection teams—depending on the team size—could be established. When faced with the inevitable choice of either inspecting a small number of versions or reducing the inspection team size, the experiment administrators chose to form five teams of reduced size (hereafter referred to as teams A, B, C, D, and E). Teams A, C, and D consisted of three members, whereas teams B and E had two and four members each, respectively. Team B consisted of two graduate students majoring in electrical engineering, both with four years of graduate education. The average years of graduate education for other teams varied from 1.3 years to 2.3 years.

The level of technical expertise possessed by the participants in the Conflict2 experiment and the Conflict1 experiment were comparable in that they were 'trained novices' as opposed to seasoned professionals. Shimeall [13] defined a trained novice as 'someone who had received an undergraduate education in computer science, had received classroom introduction to the techniques involved, and had no previous practical experience in using them'. Under a strict application of his definition, the members of all the inspection teams except team B were trained novices.

The experiment design allowed each team to select randomly and exclusively one of the eight Conflict versions by blind selection of a version number. Data on the size of implementations and the number of faults detected in the Conflict1 experiment were withheld from the teams when the selections were made. Consequently, versions 2, 3, 5, 6, and 8 were selected, and the team names, A through E, were assigned based on the version number. The team that selected the lowest-numbered version was named team A, and the team with the next lowest numbered version was named team B, and so on.

During an overview phase, the participants were briefed for two hours on the purpose of the experiment, the Fagan inspection method, the requirements specification from the Conflict1 experiment, and the guidelines to follow during inspection meetings. The participants were given a checklist of possible errors to be considered, taken from a classic software testing book [22]. Each inspection team tried to adhere fully to the principles and recommendations of the Fagan inspection techniques. For example, no inspection meetings lasted for more than two hours, and at most two inspection sessions were scheduled each day. Furthermore, the experiment administrators recommended to each team not to conduct a formal inspection meeting unless the moderator felt

that the team members were adequately prepared. However, no other restrictions were placed on the inspection teams.

3.3. Findings of Conflict2

Table II illustrates how much time each team spent on the Fagan inspection, the inspection rate, the number of major and minor faults detected, and how many mistakes (such as incorrect fault identification or classification) were made during inspections. Table I does not include minor faults, as these were not identified during the Conflict1 experiment. The average inspection rate is 210 lines of source code per hour. This rate is similar to the data, varying from 150 to 250 lines of code inspected per hour, reported on industrial inspections [23–25]. *Major faults* refer to the ones that would cause production of incorrect output, while *minor faults* refer to those that would have no impact on the correctness of the output, but could influence other aspects of software quality such as maintainability, performance, etc.

There were 179 known major faults in the five versions detected in the Conflict1 experiment, as reported in [14], and the inspection method detected 62 major faults. The number of known major faults shown in column 5 of Table II refers only to the number of faults detected in the Conflict1 experiment. In the Conflict1 experiment, only major faults were reported. The number of faults shown in Tables I and II need not be identical for data to be consistent. In Table I, the total fault detection is the sum of the number of faults detected by each method in that version, without allowing for duplicate detection. In Table II, the number of known faults is adjusted for duplicate fault detection. For example, in version 2, there were a total of 35 faults detected in the Conflict1 experiment, and six faults were detected by more than one of the five methods compared. Therefore, Table II reports that there were 29 known major faults in version 2. However, it is possible, even likely, that these versions still contain residual faults. In fact, as reported in Section 4, 56 of the 62 faults detected by inspection teams were previously unknown. As expected, the inspection teams made several mistakes. For example, several blocks of correct code were reported as being incorrect. Additionally, there were many instances where the classification of fault severity (major versus minor) was incorrect. The totals for major and minor faults in Table II (as given in columns 6 and 7) do not include the faults that were classified incorrectly (as given in columns 9 and 10).

The performance of inspection teams A and B is low in terms of both the number of faults detected and the ratio of detected major faults to the number of major faults known to exist in each Conflict version. Minor faults were not taken into consideration in evaluating the team's inspection performance, since minor faults do not result in the generation of incorrect output. Several factors seem to have been at work. The first factor is the difference between each individual team's ability. Inspection, though systematic in process, is essentially an intensive group review, and one must expect to see variation in performance. In all controlled experiments in empirical software engineering (including the LIP experiments), there were significant variations among the teams in their ability to detect faults. Teams C and D, while detecting a larger number of faults in the version than teams A and B, made more mistakes during the inspection process. Another factor seems to be the team members' technical backgrounds. The members of team B, who studied electrical engineering in their undergraduate and graduate education, were likely to lack sufficient software development experience with size and complexity exceeding toy-size examples, unlike students majoring in computer science. Since they are less likely to have personally struggled with software faults, their inspection sessions

Table II. Summary of Fagan inspections.

Team/ version	Size (LoC)	Inspection		Known major faults	Inspection detection		Inspection mistakes		
		Hours	(LoC/h)		Major faults	Minor faults	Wrong (incorrect diagnosis)	Major faults classified as minor	Minor faults classified as major
A/2	1540	5.8	266	29	7	40	1	0	9
B/3	1201	4.3	279	42	4	5	1	3	3
C/5	1544	10.2	151	40	21	15	8	6	1
D/6	2206	13.3	166	22	14	16	8	4	1
E/8	1978	10.4	190	46	16	49	1	2	5
Total		44		179	62	125	19	15	19
Average	1694	8.8	210	35.8	12.4	25	3.8	3	3.8

might not have been as intensive as those of the other teams. Another factor is the inspection speed. Although the inspection rate of teams A and B is not significantly above the reported industrial average, it is higher than that of teams C, D, and E, who detected more faults. A correlation test found that the rate of inspection was related to the number of faults detected. (Correlation coefficient is -0.95, $P \leqslant 0.05$.) That is, more time spent on the inspection generally resulted in detecting more faults. Therefore, a rapid inspection, exceeding recommended guidelines, contributed to detecting a smaller number of major faults in the versions.

In addition to the numbers of faults detected, the experiment examined the types of faults detected by the inspection teams. Shimeall [14] classified the faults found in the Conflict versions into 13 groups. Group 1 involves overly restrictive input checks (e.g., requiring all weather to move northwest only). Group 2 faults are loop conditions that are incorrectly formulated (e.g., infinite loops). Group 3 faults perform incorrect calculation due to incorrectly formulated expressions (e.g., multiplying instead of adding). Group 4 deals with uninitialized variables. Group 5 refers to the faults involving incorrect variable usage, such as substituting one array subscript with another. Faults of Group 6 lead to illegal behaviour such as division by zero failures. Group 7 deals with incorrectly formulated branch conditions. Group 8 refers to the faults involving missing branches, including both conditionals and their embedded statements. Group 9, similar to Group 8, involves missing threads (e.g., where a series of statements are missing across a section of source code, such as failure to calculate a series of results needed to update a variable). Group 10 faults involve large missing functionality where code fails to handle a particular set of requirements. (A Group 10 fault would be omitting all the required code to simulate weapon fire. A small missing functionality would be classified as either Group 8, if localized, or Group 9, if distributed through the source code.) Group 11 faults involve incorrectly ordered actions such as failure to update a value before its new value is needed. Group 12 faults refer to the cases where the formal and actual parameters are not consistently ordered. Finally, Group 13 faults involve bad data

Table III. The classification of major faults detected by inspection.

	Version 2/ team A	Version 3/ team B	Version 5/ team C	Version 6/ team D	Version 8/ team E	Group total
Group 1		0 of 2	0 of 6	0 of 1		0 of 9
Group 2			0 of 1	0 of 1	0 of 4	0 of 6
Group 3	0 of 7	1 of 8	1 of 5	0 of 7	2 of 7	4 of 34
Group 4		2 of 3	0 of 1	2 of 1	0 of 1	4 of 6
Group 5		0 of 3	0 of 4	1 of 1	1 of 1	2 of 9
Group 6	3 of 15	0 of 16	11 of 15	3 of 3	3 of 25	20 of 74
Group 7	1 of 2	1 of 2	7 of 1	2 of 1	2 of 0	13 of 6
Group 8	0 of 2	0 of 4	2 of 2	0 of 1	8 of 4	10 of 13
Group 9	2 of 1		0 of 2	1 of 0		3 of 3
Group 10		0 of 2				0 of 2
Group 11			0 of 1	5 of 0		5 of 1
Group 12				0 of 1		0 of 1
Group 13	1 of 2	0 of 2	0 of 2	0 of 5	0 of 4	1 of 15
Version total	7 of 29	4 of 42	21 of 40	14 of 22	16 of 46	62 of 179

manipulation where code causes inconsistencies in the data structure it uses (e.g., a linear linked list erroneously goes circular).

Table III classifies 62 faults according to their type as detected in the Conflict2 experiment. Data shown in the last row, 'Version total', are identical to the ones shown in Table II. It illustrates that, for example, team A was not able to detect any of the Group 3 faults although Conflict version 2 was known to contain seven such faults from the Conflict1 experiment. Likewise, for Group 6 faults, team A was able to detect only three faults although 15 faults had been detected during the Conflict1 experiment. However, it should be emphasized that the data in Table III report only the type and number of known and detected faults. That is, the three faults belonging to Group 6 and detected by team A are not necessarily a subset of the 15 faults previously known to exist in that version. This is particularly striking in Groups 7 and 11, where the number of faults detected in the Conflict2 experiment exceed those known to exist from the Conflict1 experiment. The data in Table III also show that inspection results tended to omit some groups, failed to detect all faults of a group, and tended to be clustered around certain groups. Moreover, the inspections failed entirely to detect certain types of faults known to exist in each Conflict version. For example, there were nine Group 1 faults, and teams B, C, and D failed to detect any of them. Similarly, there were 15 Group 13 faults, and four of the five inspection teams failed to detect any of them.

Inspection teams were able to detect only some of the same types of major faults known to exist in the version. There were 108 Group 3 and Group 6 faults, and inspection teams were able to detect only 24 of them. Partial detection of faults of the same type may have been caused by the fact that inspection was carried out over several sessions and on different days. Therefore, inspection teams may have forgotten about some types of faults detected in earlier sessions. These data suggest that a

Table IV. Number of major faults detected by each method (parentheses indicate the number of faults detected by no other method).

	Version 2	Version 3	Version 5	Version 6	Version 8	Method total
Voting	11(9)	19(12)	21(14)	15(8)	17(11)	83(54)
Testing	15(10)	19(16)	22(11)	10(0)	11(10)	77(47)
Self-checking	4(3)	5(1)	11(2)	6(2)	5(4)	31(12)
Code reading	5(1)	6(4)	1(0)	1(1)	19(15)	32(21)
Data-flow analysis	0(0)	2(0)	0(0)	1(1)	0(0)	3(1)
Fagan inspection	7(6)	4(1)	21(20)	14(14)	16(15)	62(56)

better inspection support environment is needed not only to broaden the types of faults detected, but also to increase the likelihood that faults similar to those previously detected faults will be reported.

The types of faults detected by inspection teams tend to be narrowly grouped. This pattern was observed for all the teams. For example, team C, which found the most faults, detected faults from only four groups, although the version contained faults belonging to 11 groups. Furthermore, more than half of the faults detected by the inspection teams belong to Groups 6, 7, and 8. As for individual teams, the majority of faults are covered by counting the faults belonging to the two most frequently detected groups. For example, for team A, five out of seven faults belong to Groups 6 and 9. Similarly, for team C, 18 out of 21 faults belong to the Groups 6 and 7. Likewise, eight out of 14 (for team D) and 11 out of 16 (for team E) faults belong to two groups.

4. COMPARISON OF SIX FAULT DETECTION METHODS

This section compares the results of the Conflict2 experiment to those of the Conflict1 experiment. The number of faults detected by each of the six methods is determined by combining results from Conflict1 and Conflict2. The type of faults is discussed using the fault grouping classification described in Table III. An examination of the average amount of effort required to detect a fault introduces weight factors to facilitate objective comparison of manual and automated methods. The section closes by providing a qualitative comparison of the degree of detail in fault detection provided by each method.

4.1. Number of faults detected

Table IV illustrates the number of major faults detected by each method. While the majority of data in Table IV are derived from data published from Conflict1 and cited in Table I, some explanations are in order. As shown in the lower right corner of the table, 56 of the 62 major faults detected by inspection teams had not been detected previously. This confirms the discussion of fault detection related to Table III. Of the remaining six faults, four were detected only by the testing method,

Table V. Type of faults detected by each method.

	Voting	Testing	Fagan inspection	Self-checks	Code reading	Data-flow analysis
Group 1	1	6	0	9	1	0
Group 2	2	1	0	4	0	0
Group 3	33	5	4	5	0	0
Group 4	3	1	4	1	1	3
Group 5	8	2	2	0	2	0
Group 6	10	49	20	4	19	0
Group 7	3	2	13	0	3	0
Group 8	9	4	10	1	5	0
Group 9	2	1	3	1	0	0
Group 10	0	2	0	0	0	0
Group 11	1	0	5	0	0	0
Group 12	1	1	0	0	0	0
Group 13	10	3	1	6	1	0
Total	83	77	62	31	32	3

voting, code reading, and data-flow analysis. The other two faults were detected by more than one method (i.e. voting/code reading and voting/data-flow analysis). For example, a fault in version 2 that had previously been detected only by the code reading technique was also detected by inspection. Therefore, data shown in the corresponding cell in Table I, 5(2), have been changed to an updated value, 5(1). Since six of the seven faults that had been detected by inspection teams in version 2 had not been previously detected, the value 7(6) is assigned in Table IV.

Comparison of the number of major faults alone indicates that the Fagan inspection detected fewer faults than did the voting or the testing method. While such raw data might create an impression that Fagan inspection is inferior to voting or the testing method, further analysis reveals that these methods are complementary to each other. There were 288 major faults detected by some combination of the six methods, and 191 faults (or 66.3%) were detected by only one method. In a limited comparison of just the three methods (voting, the testing method, and Fagan inspection) that detected a significantly larger number of faults than the three other methods (code reading, data-flow analysis, and self checking), 157 out of 222 (or 70.7%) faults were detected by only one method.

4.2. Type of faults detected

Comparison of the types of faults detected by each method, shown in Table V, confirms that voting, the testing method, and Fagan inspection methods are largely complementary to each other. While it is true that each method was able to detect various types of faults, voting was shown to be the most effective in detecting faults involving calculation faults, such as the use of the wrong expression in calculation. For example, where the testing method and inspection detected a small number of Group 3 faults,

Table VI. Average number of hours needed to detect a fault.

Methods	Computer hours	Human hours	Major faults per version	Average hours to detect a fault with weight factors (estimated)	
				10	50
Code reading	0	36	5.6	*6.4*	*6.4*
Data-flow analysis	40	1	0.6	*8.3*	*3.0*
Testing	84	373	15.4	*24.8*	*24.3*
Voting	1415	6	16.2	*9.1*	*2.1*
Fagan inspection	0	47.11	12.4	*3.8*	*3.8*
Self-checks	No concrete data available since self-checks were added during development				

voting detected 33 such faults. This observation seems reasonable; faults in formulating arithmetic expressions are most likely to result in an incorrect calculation being carried out. Ultimately, this results in the generation of outputs with differing values, and the voting mechanism can easily detect mismatches. On the other hand, the testing method was more effective in detecting Group 6 faults, which cause illegal behaviour such as division by zero. As for inspection, it seems better at detecting faults involving incorrect or missing branch conditions, belonging to Groups 7 and 8, than does voting or the testing method.

The data shown in Table V also indicate that the Fagan inspection method is superior to code reading by stepwise refinement. Inspection teams were able to detect larger numbers (i.e. 62 versus 32 faults) and more diverse types of faults than code reading did. Although the two methods share similarities, increased levels of rigour and group collaboration seem to be the contributing factors to better detection by Fagan inspections.

4.3. Effort of fault detection

In order to perform a quantitative comparison of the cost-effectiveness of various methods, weight factors were introduced. There is no widely accepted measure comparing personnel costs and hardware costs. In Table VI, illustrative weights of 10 and 50 computer hours[¶] are counted as being equivalent to one human hour. This comparison assumes that it costs about $100 000 a year, including overheads, for a company to hire a technical staff member. Under an assumption that a workstation costing approximately $30 000 is to be amortized over three years, the yearly cost of a workstation is about $10 000 a year resulting in a weight factor of 10. Replacing a workstation with a $2000 PC becoming

[¶]These figures are selected as illustrative—useful further analysis may incorporate the perhaps-substantial costs of writing software, installing software, hardware and software repairs, etc. The estimation of these costs was not performed due to the large number of variables involved (including size and configuration issues).

'obsolete' in a year, the weight factor becomes 50. With impressive and continual reduction in PC prices, it seems feasible that a larger number should be used as a proper weight factor in the future. It should be noted that the cost figure, used as an example to illustrate the general trend, is not limited only to the hardware cost. The figure should be interpreted to include the cost of system software such as the operating system as well as custom software needed to carry out N-version voting. While it is true that developing customized software is an expensive endeavour, software needed to carry out mechanical voting is relatively simple (e.g., similar to the `diff` utility in Unix). However, it is neither the purpose nor within the scope of this study to determine the precise weight factor. This calculation would depend on several factors whose values might vary from one organization to another, and a proper value can be substituted as needed.

The cost-effectiveness of various fault detection methods is shown in Table VI. The comparison is focused on the testing method, voting, and Fagan inspection because these methods detected substantially more faults than the other techniques. Since the Conflict1 experiment collected no data on the number of hours spent in developing self-checks, this analysis excludes that method from this comparison.

The data indicate that inspection and voting methods are more cost-effective in execution than the testing method. However, the cost of developing one or more redundant versions, unaccounted for in this data, must be taken into consideration if the voting method is to be used for the sole purpose of detecting faults. In other words, unless fault-tolerance capability is explicitly called for, voting is unlikely to be as inexpensive as the raw data indicate. On the other hand, all pertinent inspection or testing costs are accounted for in this data.

4.4. Detail in fault detection

To correct a fault, one must not only know the fault exists, but also some detail as to the location and type of the fault. Therefore, in comparing detection methods, one must compare the degree of detail in the information each method produces. That is, the inspection method, code reading, and static analysis identify not only the precise location of the faults but also their types. The testing method and self-checks do this as well, but to a much lesser degree of precision and with a sometimes-complex analysis required to match input cases with code locations. Voting, on the other hand, simply indicates that the comparison of outputs did not match, and provides no further detail. If two-version voting is used, it does not reveal which of the versions is incorrect or where the fault is in the version. If the three-version voting scheme is used, one might be able to determine which of the three versions produced the minority result. However, the minority result is not necessarily the incorrect result.

5. CONCLUSIONS

This paper reports the results of the Conflict2 experiment in which five of the eight versions (produced in the Conflict1 experiment) were each subjected to a Fagan inspection. The Conflict2 experiment extended the empirical comparison from five (in the Conflict1 experiment) to six methods: voting, the testing method used in the Conflict1 experiment, Fagan inspection, code reading by stepwise refinement, self-checks, and data-flow analysis. This comparison examined relative fault detection capability as well as relative cost-effectiveness. The primary findings of the study can be summarized as follows. First, voting, the testing method, and Fagan inspection detected more faults than the

other methods. Second, voting, the testing method, and the Fagan inspection method are largely complementary to each other in that a significant portion of the faults detected by each method were not detected by the other two methods. Third, the Fagan inspection method is more cost-effective than the testing method. Issues involving the comparison of manual and automated effort prevented the determination of definite and quantitative conclusions of the relative cost-effectiveness of the voting and inspection methods in this experiment. However, it does not seem practical to develop multiple versions for the sole purpose of detecting faults in the program, rather than using multiple versions to satisfy a requirement for providing software fault-tolerance capability. Therefore, this study concludes that the testing method and Fagan inspection methods comprise the most effective combination of the fault detection methods studied.

Detailed analysis of the type of faults detected by the Fagan inspection method yielded insight into how the effectiveness of the Fagan inspection method could be further improved. Inspection teams were able to detect faults belonging to only a small number of groups. It seems that the checklist, a generic enumeration of common programming faults [22] used in the Conflict2 experiment, may have been inadequate. The data also indicate that inspection teams were not able to detect all the occurrences of a particular fault type. It seems that inspection teams did not have convenient access to information on the types of faults they had already detected (perhaps at inspection meetings held in the previous days), and thus may have forgotten about them. An integrated inspection environment seems necessary to overcome such shortcomings. For example, the checklist could be dynamically augmented with samples of erroneous code. A dynamic testing tool could be provided to determine more accurately whether a block of code is erroneous or not. The authors plan to develop such a software tool and conduct another controlled experiment, using the Conflict versions, to measure effectiveness of tool-supported inspection methods empirically.

Finally, the importance of sharing raw data used in this study with other researchers working on empirical software engineering cannot be overemphasized. The analysis reported in this paper can be independently verified, if desired, by other researchers. More importantly, other researchers might be able to perform different types of statistical analysis on the data to obtain different or deeper insights. Progress in empirical software engineering critically depends on researchers sharing not only ideas and interpreted results but also the raw data from experiments. With the Internet becoming an integral part of society and the availability of software tools to edit and publish data on the Internet, the authors strongly believe in their responsibility to share these data with other researchers, and have published them at http://salmosa.kaist.ac.kr/research/empirical-studies/.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the insightful comments of the reviewers and the diligent work of Professors Lee White and Martin Woodward, all of whom added greatly to the quality of this paper. Ms Claire Dixon provided valuable assistance in technical editing. We also wish to acknowledge our debt to the work of Professor Nancy Leveson.

The work of Sung Deok Cha was supported, in part, by the Korea Science and Engineering Foundation (KOSEF) through the telepresence project at the Advanced Information Technology Research Center (AITrc).

Dr Shimeall's participation in this work was supported in part by funds administered by the Naval Postgraduate School Research Council and in part by funds from the Department of Defense.

REFERENCES

1. Gray J. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability* 1990; **39**(4):409–418.
2. Beizer B. *Software Testing Techniques* (2nd edn). Van Nostrand Reinhold: New York, NY, 1990.
3. Chen L, Avizienis A. N-version programming: A fault tolerance approach to the reliability of software. *Proceedings of the 8th International Symposium on Fault-Tolerance Computing*. IEEE Computer Society Press: Los Alamitos, CA, June 1978; 3–9.
4. Fagan ME. Design and code inspections to reduce errors in programming development. *IBM Systems Journal* 1976; **5**(3):182–211.
5. Fagan ME. Advances in software inspection. *IEEE Transactions on Software Engineering* 1986; **12**(7):744–751.
6. Myers GJ. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM* 1978; **21**(9):760–768.
7. Floyd RW. Assigning meanings to programs. *Proceedings of the Symposium on Applied Mathematics*, vol. XIX. American Mathematical Society: New York, April 1967; 19–32.
8. Rosenblum DS. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering* 1995; **21**(1):19–31.
9. Fosdick LD, Osterweil LJ. Data flow analysis in software reliability. *ACM Computing Surveys* 1976; **8**(3):305–330.
10. Leveson NG, Cha SS, Knight JC, Shimeall TJ. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering* 1990; **16**(4):432–443.
11. So S, Lim Y, Cha SD, Kwon YR. An empirical study on software error detection: Voting, instrumentation, and Fagan inspection. *Proceedings of the Asia–Pacific Software Engineering Conference*. IEEE Computer Society Press: Los Alamitos, CA, 1995; 345–351.
12. Knight JC, Leveson NG. An experimental evaluation of the assumption of independence in multiprogramming. *IEEE Transactions on Software Engineering* 1986; **12**(1):96–109.
13. Shimeall TJ, Leveson NG. An experimental evaluation of software fault tolerance and fault elimination. *IEEE Transactions on Software Engineering* 1991; **17**(2):173–182.
14. Shimeall T. An empirical comparison of software fault elimination and fault tolerance. *PhD Dissertation*, Department of Information and Computer Science, University of California, Irvine, 1989.
15. Anderson T, Lee PA. *Fault Tolerance: Principles and Practice*. Prentice-Hall: Englewood Cliffs, NJ, 1981.
16. Linger RC, Mills HD, Witt BL. *Structured Programming: Theory and Practice*. Addison-Wesley: Reading, MA, 1979; 147–212.
17. Frankl PG, Weyuker EJ. Data flow testing in the presence of unexecutable paths. *Proceedings of the Workshop on Software Testing*, Banff, Canada, July 1986. IEEE Computer Society Press: Los Alamitos, CA, 1996; 4–13.
18. Wheeler DA, Brykczynksi B, Meeson RN (eds.). *Software Inspection: An Industry Best Practice*. IEEE Computer Society Press: Los Alamitos, CA, 1996.
19. Bisant DB, Lyle JR. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering* 1989; **15**(10):1294–1304.
20. Votta LG. Does every inspection need a meeting? *ACM SIGSOFT '93-Proceedings of the 1st ACM SIGSOFT Symposium on Software Development Engineering*. Association for Computing Machinery: New York, 1993; 107–114.
21. Porter AA, Siy HP, Mockus A, Votta LG. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology* 1998; **7**(1):41–79.
22. Myers GJ. *The Art of Software Testing*. John Wiley: New York, NY, 1979, pp. 30.
23. Ebenau RG. Predictive quality control with software inspection. *Software Inspection: An Industry Best Practice*, Wheeler DA, Brykczynksi B, Meeson RN (eds.). IEEE Computer Society Press: Los Alamitos, CA, 1996; 147–156.
24. Weller EF. Lessons from three years of inspection data. *IEEE Software* 1993; **10**(5):38–45.
25. Barnard J, Price A. Managing code inspection information. *IEEE Software* 1994; **11**(2):56–69.

