An Empirical Study on Testing and Fault Tolerance for Software Reliability Engineering

Abstract

Software testing and software fault tolerance are two major techniques for developing reliable software systems, yet limited empirical data are available in the literature to evaluate their effectiveness. We conducted a major experiment to engage 34 programming teams to independently develop multiple software versions for an industry-scale critical flight application, and collected faults detected in these program versions. To evaluate the effectiveness of software testing and software fault tolerance, mutants were created by injecting real faults occurred in the development stage. The nature, manifestation, detection, and correlation of these faults were carefully investigated. The results show that coverage testing is generally an effective mean to detecting software faults, but the effectiveness of testing coverage is not equivalent to that of mutation coverage, which is a more truthful indicator of testing quality. We also found that exact faults found among versions are very limited. This result supports software fault tolerance by design diversity as a creditable approach for software reliability engineering. Finally we conducted domain analysis approach for test case generation, and concluded that it is a promising technique for software testing purpose.

1. Introduction

Fault removal and fault tolerance are two major approaches in software reliability engineering [1]. Fault removal techniques detect and remove software faults during software development so that they will not be present in the final product, while fault tolerance techniques detect and tolerate software faults during software operation so that they will not interrupt the service delivery.

The main fault removal technique is software testing. The key issue in software testing is the selection of test cases and the evaluation of testing effectiveness. Two major schemes in test case selection and evaluation are data flow coverage testing [2] and mutation testing [3].

Data flow coverage is a technique to provide measure of test sets and test completeness by executing the test cases and measuring how program codes are exercised. Some studies show the high data flow coverage brings high software reliability [4]. The observation of a correlation between good data flow testing and a low field fault rate is reported for the usefulness of data flow coverage testing [5, 6]. Impact of test coverage to fault detection is also performed [7]. Furthermore, research efforts have been conducted to establish relationship between test coverage and software reliability [8, 9]. As most experimental investigations are “once-only” efforts, however, conclusive evidence about the effectiveness of coverage is still lacking.

The approach to mutation testing, on the other hand, begins by creating many versions of a program. Each of these versions is "mutated" to introduce a single fault. These "mutant" programs are then run against test cases with the goal of causing each faulty version to fail. Each time a test case causes a faulty version to fail, that mutant is considered "killed.” Empirical studies on mutation testing are widely performed [10, 11, 12, 13]. Mutation testing is also applied for integration testing [14] and program analysis [15]. However, in most previous investigations, mutants are artificially generated with hypothetical faults. The testing process produces an enormous number of mutants, and each mutant must be recompiled and tested. These mutants are either too trivial (too easily killed) or too unrealistic (too hard to be activated).

On the fault tolerance side, the main technique is software design diversity, including recovery blocks [16], N-version programming [17], and N self-checking programming [18]. Design diversity approach achieves fault-tolerant software systems through the independent development of program versions from a common specification. It is a software reliability engineering technique subject to continuous investigations by many researchers regarding its experimentation [19, 20, 21], modeling [22, 23, 24], and evaluation [25, 26, 27]. The effectiveness of design diversity, however, heavily depends on the failure correlation among the developed multiple program versions [28, 29, 30], which remains a debatable research issue.

Our research is motivated by the lack of real world project data for investigation on software testing and fault tolerance techniques together, with comprehensive analysis and evaluation. Subsequently we conducted a real-world project and engaged multiple programming teams to independently develop program versions based on an industry-scale avionics application. We conducted detailed experimentation to study the nature, source, type, detectability, and effect of faults uncovered in the program versions, and to learn the relationship among these faults and the correlation of their resulting failures. We applied the mutation testing techniques to reproduce mutants with *real* faults, and investigated the effectiveness of data flow coverage, mutation coverage, and design diversity for fault coverage. From the results, we examined different hypotheses on software testing and fault tolerance schemes, and drew a number of interesting observations. Finally, we performed a new software test case generation technique [31] based on domain analysis approach [32] and evaluated its effectiveness.

2. Project Descriptions and the Experimental Procedure

In the spring of 2002 we formed 34 independent programming teams at the Chinese University of Hong Kong to design, code, test, evaluate, and document a critical application taken from industry. Each team was composed of 4 senior-level undergraduate Computer Science students for a 12-week long project in a software engineering course. We portray the project details, the software development procedure and the creation of mutants with the faults uncovered during software testing phase. Setup for the evaluation test environment and the initial metrics are also described.

2.1 RSDIMU Project

The specifications of a critical avionics instrument, Redundant Strapped-Down Inertial Measurement Unit (*RSDIMU*), were used in our project investigation. RSDIMU was first engaged in [33] for a NASA-sponsored 4-university multi-version software experiment. It is part of the navigation system in an aircraft or spacecraft. In this application, developers are required to estimate the vehicle acceleration using the eight accelerometers mounted on the four triangular faces of a semi-octahedron in the vehicle. As the system itself is fault tolerant, it allows the calculation of the acceleration when some of the accelerometers fail. Figure 1 show the system data flow diagram.

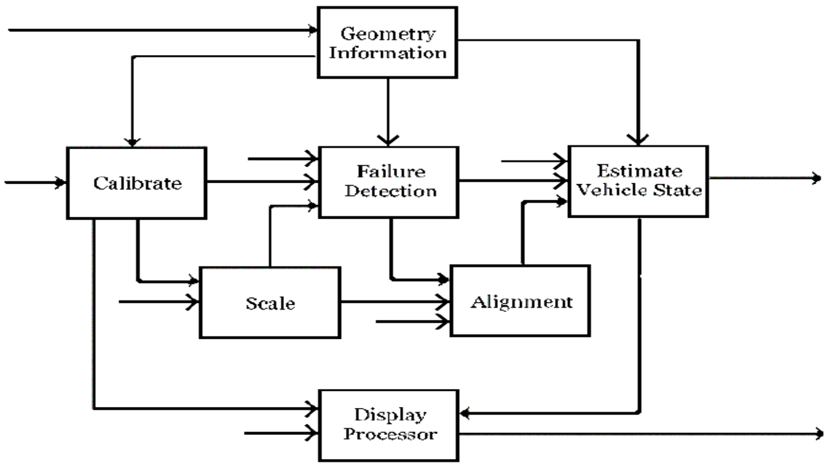

Figure 1 RSDIMU System Data Flow Diagram

The accelerometer measures specific force along its associated measurement axis where specific force is the difference between the RSDIMU's inertial linear acceleration and the acceleration due to gravity. There are two kinds of input processing. The first type is the information describing the system geometry ("Geometry Information"). The second type is the accelerometer readings from the accelerometers, which need to be pre-processed through calibration ("Calibrate") and scaling ("Scale").

The program should perform two major functions. First is to conduct a consistency check to detect and isolate failed accelerometers ("Failure Detection"). The second is to use the accelerometers found to be good by the first check to provide estimates of the vehicle's linear acceleration expressed as components along different alignments ("Alignment" and "Estimate Vehicle State").

For output processing, the primary outputs are the accelerometer status vector specifying either a failed or an operational mode ("Failure Detection"), and a set of estimates for the vehicle's linear acceleration based on various subsets of the operational accelerometers ("Estimate Vehicle State"). The secondary output is the information which drives a display panel and provides system status ("Display Processor").

2.2 Software Development Procedure

The waterfall model was applied in this software development project. Six phases were conducted in the development process:

Phase 1: Initial design document (duration: 3 weeks)

The purpose was to allow the programmers to get familiar with the specifications, so as to design a solution to the problem. At the end of this phase, each team delivered a preliminary design document, which followed specific guidelines and formats for documentation.

Phase 2: Final design document (duration: 3 weeks)

The purpose was to let each team obtain some feedback from the coordinator to adjust, consolidate, and complete their final design. Each team was also requested to conduct at least one design walkthrough. At the end of this phase, each team delivered (1) a detailed design document, and (2) a design walkthrough report.

Phase 3: Initial code (duration: 1.5 weeks)

By the end of this phase, programmers finished coding, conducted a code walkthrough, and delivered the initial, compliable code in the C language. Each team was required to use the RCS revision control tool for configuration management of the program modules.

Phase 4: Code passing unit test (duration: 2 weeks)

Each team was supplied with sample test data sets for each module to check the basic functionalities of the module. They were also required to build their own test harness for the testing purpose.

Phase 5: Code passing integration test (duration: 1 week)

Several sets of test data were provided to each programming team for integration testing. This testing phase was aimed to guarantee that the software was suitable for testing as an integration system.

Phase 6: Code passing acceptance test (duration: 1.5 weeks)

Programmers formally submitted their programs for a stringent acceptance test, where 1200 test cases were used to validate the final code. At the end of this phase all 34 teams passed the acceptance test. It is noted, that the requirement for this acceptance test was the same as the operational test conducted in [33], which was much tougher than the original acceptance test in [33].

2.3 Mutant creation

RCS was required for source control for each team. Every code change of each program file at each check-in can therefore be identified. Software faults found during each stage are also identified. These faults were then injected into the final program versions to create mutants, each contain one programming fault. We selected 21 program versions for detailed investigation, and created 426 mutants. We disqualified the other 13 versions as their developers did not follow the development and coding standards which were necessary for generating meaningful mutants from their projects.

The following rules are applied in the mutant creation process:

1. Low-grade errors, for example compilation error and core dump exception, are not created.

2. Some changes were only available in middle versions. For example, the changes between 1.1 and 1.2 may not be completely identified in the final version. These changes are then ignored.

3. Code changes for debugging purposes are not included.

4. Modifications of the function prototypes are excluded.

5. As the specification does not mention about memory leaks, mutants are not created for any faults leading to memory leaks.

6. The same programming error may span in many blocks of code. For example: a vector missed the division by 1000.0 may occur everywhere in a source file. It is counted as a single fault.

2.4 Setup of Evaluation Test

In order to evaluate the effectiveness of data flow testing schemes, we set up an evaluation test environment. We employed the ATAC (Automatic Test Analysis for C) [6, 34] tool to analyze and compare coverage of testing conducted in the 21 program versions, together with their 426 mutants. For each round of evaluation test, all 1200 acceptance test cases were exercised on these mutants. This was a very intensive testing procedure, as all the resulting failures from each mutant were analyzed, their coverage measured, and cross-mutant failure results compared.

60 Sun machines running Solaris were involved in the evaluation test. The evaluation test script run on a master host, and distributed each mutant as a running task to another machine. The execution results were collected in network file systems (NFS). One cycle of evaluation test took 30 hours, and the test results generated around 20GB of a total of 1.6 million files

2.5 Program Metrics

Table 1 shows the program metrics for the 21 versions engaged in the evaluation test, and the mutants each of them generated. It can be noted that the size of these programs varies from 1455 to 4512 source lines of code. Each version produced a number of mutants ranging from 9 to 31. The data flow metrics are also listed in Table 1.

3. Static Analysis of Mutants: Fault Classification and Distribution

Judging from the number of programming teams involved and the quantify of mutants generated, this investigation is probably the largest scale experiment in the literature regarding injecting actual programming faults in real-world software application for multiple program versions. We first perform static analysis of the mutants regarding their defect type, qualifier, severity, development stage occurrence and effect code lines. Note we use "defect" and "fault" interchangeably.

3.1 Mutant Defect Type Distribution

Each mutant is assigned with a defect type according to [35]. The statistics is show in Table 2.

3.2 Mutant Qualifier Distribution

Each mutant is assigned with a qualifier. The statistics is show in Table 3, with the following definitions:

- Incorrect – The defect was a mistake in computing. For example: typo, wrong algorithm, etc.
- Missing – Something was missing to cause the defect.
- Extraneous – Useless addition caused the error.

Id	Lines	Modules	Functions	Blocks	Decisions	C-Use	P-Use	Mutants
01	1628	9	70	1327	606	1012	1384	25
02	2361	11	37	1592	809	2022	1714	21
03	2331	8	51	1081	548	899	1070	17
04	1749	7	39	1183	647	646	1339	24
05	2623	7	40	2460	960	2434	1853	26
07	2918	11	35	2686	917	2815	1792	19
08	2154	9	57	1429	585	1470	1293	17
09	2161	9	56	1663	666	2022	1979	20
12	2559	8	46	1308	551	1204	1201	31
15	1849	8	47	1736	732	1645	1448	29
17	1768	9	58	1310	655	1014	1328	17
18	2177	6	69	1635	686	1138	1251	10
20	1807	9	60	1531	782	1512	1735	18
22	3253	7	68	2403	1076	2907	2335	23
24	2131	8	90	1890	706	1586	1805	9
26	4512	20	45	2144	1238	2404	4461	22
27	1455	9	21	1327	622	1114	1364	15
29	1627	8	43	1710	506	1539	833	24
31	1914	12	24	1601	827	1075	1617	23
32	1919	8	41	1807	974	1649	2132	20
33	2022	7	27	1880	1009	2574	2887	16
Average	2234.2	9.0	48.8	1700.1	766.8	1651.5	1753.4	Total: 426

Table 1 Program metrics for 21 versions

Defect types	Number	Percent
Assign/Init:	136	31%
Function/Class/Object:	144	33%
Algorithm/Method:	81	19%
Checking:	60	14%
Interface/OO Messages	5	1%

Table 2 Defect Type Distribution

Qualifier	Number	Percent
Incorrect:	267	63%
Missing:	141	33%
Extraneous:	18	4%

Table 3 Qualifier Distribution

3.3 Mutant Severity Distribution

The severity distribution according to the following definitions is listed in Table 4.

A Level (Critical): If the mutant could not generate final result (in this project, it's the acceleration value) due to the fault.

B Level (High): If the mutant generated wrong final result due to the fault.

C Level (Low): If the mutant generated the correct final result but produced some other incorrect output (for example, the display results were erroneous.)

D Level (Zero): If the mutant passed all test cases but failed for some special minor reason (for example, incorrect voting sequence without affecting out values.)

Note that in Table 4, "Highest Severity" records the highest level of severity among all failed test cases for a mutant, while "First Failure Severity" records the failure severity at the first time when a failure occurred to the mutant.

3.4 Fault Distribution over Development Stage

The sources of faults came from different stages of the development. This distribution is shown in Table 5.

3.5 Mutant Effect Code Lines

The number of code lines span affected by each mutant was measured by manual inspection. Table 6 lists the details. In previous research efforts on mutation testing, usually the faults were artificially injected which simple code changes such as the replacement of a logic operator in a conditional statement or the modification of a operand value, and the code line span was limited to one or a few lines. It can be seen from Table 6 that in our experiment, an average 11.39 code lines were affected by a fault, truthfully reflecting the reality.

Severity Level	Highest Severity		First Failure Severity	
	Number	Percentage	Number	Percentage
A Level (Critical):	12	2.8%	3	0.7%
B Level (High):	276	64.8%	317	74.4%
C Level (Low):	95	22.3%	99	23.2%
D Level (Zero):	43	10.1%	7	1.6%

Table 4 Severity Distribution

Stage	Number	Percentage
Init Code	237	55.6%
Unit Test	120	28.2%
Integration Test	31	7.3%
Acceptance Test	38	8.9%

Table 5 Development Stage Distribution

Lines	Number	Percent
1 line:	116	27.23%
2-5 lines:	130	30.52%
6-10 lines:	61	14.32%
11-20 lines:	43	10.09%
21-50 lines:	53	12.44%
>51 lines:	23	5.40%
Average	11.39	

Table 6 Fault Effect Code Lines

4. Dynamic Analysis of Mutants: Effects on Software Testing and Fault Tolerance

The test cases conducted in the evaluation test is described in Table 7. Based on execution of these test cases over the mutants, we analyzed fault and failure relationship. We examined the effectiveness of the test cases by their test coverage measures, and their ability to kill the mutants. We also studied the fault detecting capability of each test case, and obtained the non-redundant set of test cases which can cover all mutants.

4.1 Effectiveness of Code Coverage

In order to answer the question whether testing coverage is an effective means for fault detection, we executed the 426 mutants over all test cases and observed whether additional coverage of the code was achieved when the mutants were killed by a new test case. In the experiment, we excluded the mutants which failed upon the first test case, as we wanted to take a more conservative view in evaluating test coverage by analyzing only those mutants which passed at least the first test case and then failed in later cases. There were a total of 252 mutants included in this analysis.

Effectiveness of testing coverage in revealing faults is shown in Table 8. Here we use the common test coverage measures: block coverage, decision coverage, C-use coverage and P-use coverage [2, 36]. The second to fifth column identify the number of faults in relation to changes of blocks, decision, c-uses and p-uses, respectively. For example, “6/11” for version ID “1” under the “Blocks” column means during the evaluation test stage, six out of eleven faults in program version 1 showed the property that when these faults were detected by a test case, block coverage of the code increased. On the other hand, five faults of program version 1 were detected by test cases without increasing the block coverage. The last column “Any” counts the total number of mutants whose coverage increased in any of the four coverage measures when the mutants were killed.

The result clearly shows the increase in coverage is closely related to more fault detections. Out of 252 mutants under analysis, 155 of them show some kinds of coverage increase when they were killed. This represents a high ratio of 61.5%. The range, however, is very wide (from 22.2% to 94.7%) among different versions. This indicates programmer's individual capability accounted for a large variety in the faults they created and the detectability of these faults.

One may hypothesize that when there are more (or less) faults in a program version, it may be easier (or more difficult) to detect these faults with coverage-based testing schemes. A plot of the number of mutants against effective percentage of coverage is therefore obtained in Figure 2. It can be seen in Figure 2 that the number of mutants in each version (i.e., the number of faults in the program) can not indicate one way or the other the effectiveness of test coverage in exploring the faults (by killing the mutants).

Case ID	Description of the test cases.
1	A fundamental test case to test basic functions.
2-7	Test cases checking vote control in different order.
8	General test case based on test case 1 with different display mode.
9-19	Test varying valid and boundary display mode.
20-27	Test cases for lower order bits.
28-52	Test cases for display and sensor failure.
53-85	Test random display mode and noise in calibration.
87-110	Test correct use of variable and sensitivity of the calibration procedure.
86, 111-149	Test on input, noise and edge vector failures.
150-151	Test various and large angle value.
152-392	Test cases checking for the minimal sensor noise levels for failure declaration.
393-800	Test cases with various combinations of sensors failed on input and up to one additional sensor failed in the edge vector test.
801-1000	Random test cases. Initial random seed for 1st 100 cases is: 777, for 2nd 100 cases is: 1234567890
1001-1200	Random test cases. Initial random seed is: 987654321 for 200 cases.

Table 7 Test Case Description

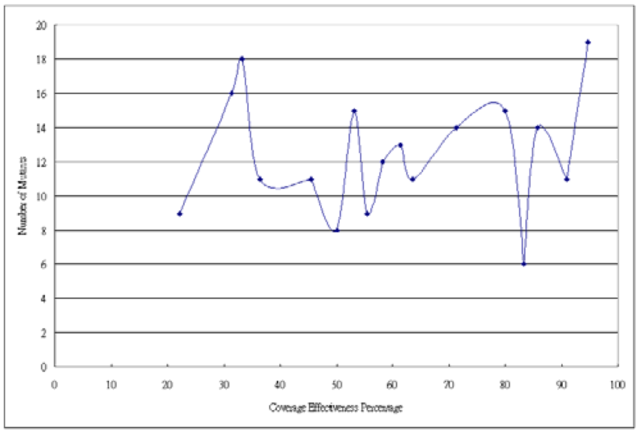

Figure 2 Relations between Numbers of Mutants against Effective Percentage of Coverage

Version ID	Blocks	Decisions	C-Use	P-Use	Any
1	6/11	6/11	6/11	7/11	7/11(63.6%)
2	9/14	9/14	9/14	10/14	10/14(71.4%)
3	4/8	4/8	3/8	4/8	4/8(50.0%)
4	7/13	8/13	8/13	8/13	8/13(61.5%)
5	7/12	7/12	5/12	7/12	7/12(58.3%)
7	5/11	5/11	5/11	5/11	5/11(45.5%)
8	1/9	2/9	2/9	2/9	2/9(22.2%)
9	7/12	7/12	7/12	7/12	7/12(58.3%)
12	10/19	17/19	11/19	17/19	18/19(94.7%)
15	6/18	6/18	6/18	6/18	6/18(33.3%)
17	5/11	5/11	5/11	5/11	5/11(45.5%)
18	5/6	5/6	5/6	5/6	5/6(83.3%)
20	9/11	10/11	8/11	10/11	10/11(90.9%)
22	12/14	12/14	12/14	12/14	12/14(85.7%)
24	5/6	5/6	5/6	5/6	5/6(83.3%)
26	2/11	4/11	4/11	4/11	4/11(36.4%)
27	4/9	5/9	4/9	5/9	5/9(55.6%)
29	10/15	10/15	11/15	10/15	12/15(80.0%)
31	7/15	7/15	7/15	7/15	8/15(53.3%)
32	3/16	4/16	5/16	5/16	5/16(31.3%)
33	7/11	7/11	9/11	10/11	10/11(90.9%)
Overall	131/252 (60.0%)	145/252 (57.5%)	137/252 (53.4%)	152/252 (60.3%)	155/252 (61.5%)

Table 8 Fault Detection Related to Changes of Test Coverage

4.2 Test Case Contribution: Test Coverage vs. Mutant Coverage

The contribution of each test case in block coverage of the total 426 mutants, measured across all executed mutants, is recorded and depicted in Figure 3. The vertical axis indicates the average percent of block coverage by each test case. Lines A, B, C, D, E represent the border for test cases 111, 152, 393, 801 and 1001, respectively. They mark the distinct boundaries of different test cases described and tabulated in Table 7. Figure 3 shows various fault detection capabilities of different kinds of test cases, as separated by the lines. The total average block coverage is 45.86%, with a range from 32.42% to 52.25%.

The decision, C-use and P-use coverage measures expose *exactly* the same pattern except for their absolute values, and thus omitted here. The overall average value of these measures is shown in Table 9.

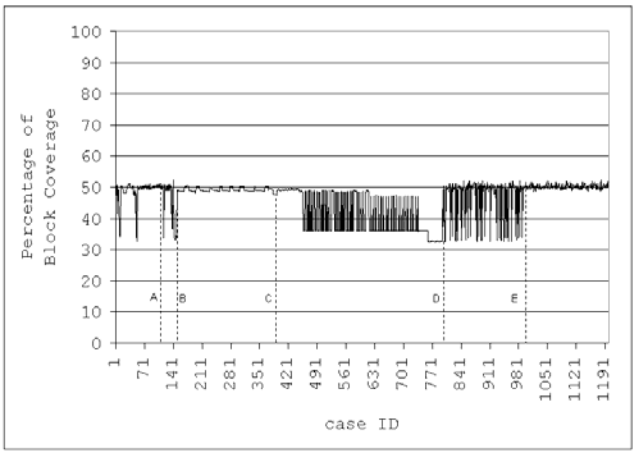

Figure 3 Test Case Contribution on Program Coverage

Percentage of Coverage	Blocks	Decision	C-Use	P-Use
Average	45.86%	29.63%	35.86%	25.61%
Maximum	52.25%	35.15%	41.65%	30.45%
Minimum	32.42%	18.90%	23.43%	16.77%

Table 9 Percentage of Test Case Coverage

The contribution of each test case in covering (killing) the mutant population is shown in Figure 4. The vertical axis represents the number of mutants that can be killed by each test case. Lines A, B, C, D, E represent again the distinct boundaries of different test cases. Similar to Figure 3, Figure 4 also clearly portrays the fault detection profiles of each kind of test case. The average number of faults detected by a test case is 248, with 163 as minimum and 334 as maximum.

The comparison between Figure 3 and Figure 4 offers profound implications: they reveal similarity and difference between *code* coverage and *fault* coverage. On the one hand, test coverage and mutant coverage show similar capability in revealing patterns in the test cases, giving credit to code coverage as a good indicator for test variety. On the other hand, the code coverage value alone is not a good indicator for test quality in terms of fault coverage. Higher and more stable code coverage, e.g., that achieved by test cases 1001-1200, may result in lower and unstable fault coverage.

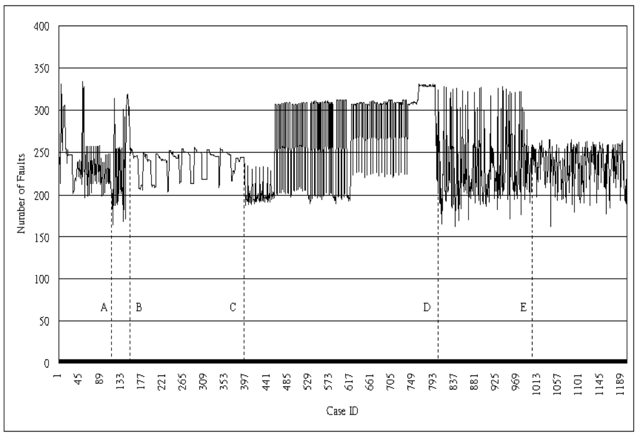

Figure 4 Test Case Contributions on Mutant Coverage

We note that this kind of quantitative analysis on test case efficiency with the injection of actual faults in real-world project has seldom been reported in the literature.

4.3 Finding Non-redundant Set of Test Cases

One important issue in software testing is the removal of redundant test cases. If two test cases kill exactly the same mutants, one of them can be regarded as redundant. By eliminating all such redundant cases, the remaining test cases constitute a non-redundant test set.

Figure 5 shows the non-redundant test set from the 1200 test cases. The gray lines indicate redundant cases, while the black blocks indicate the set of non-redundant test cases. The size of this test set is 698 test cases.

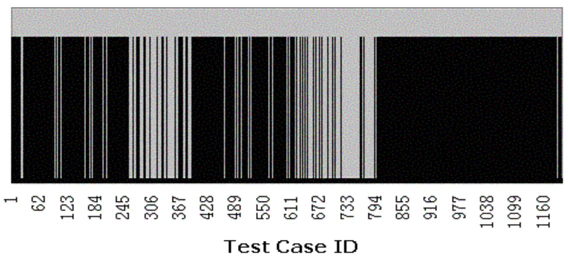

Figure 5 Non-redundant Set of Test Cases

We observe that redundant test case is rare after test case 800. In examining Table 7, we note that test cases after 800 are random test cases. They do not focus on any particular aspect of the program, thus avoiding redundancy.

4.4 Relationship between Mutants

In the interest of software fault tolerance, we also investigated fault similarity and failure correlation based on the mutant population. The test result of every success/failure test result can be collected to form a binary string of 1200 bits. Based on comparisons of the binary strings from all 426 mutants, three mutant relations can be defined:

- *Related* mutants: Two mutants have the same success/failure result on the 1200-bit binary string.
- *Similar* mutants: Two mutants have the same binary string and with the same erroneous output variables.
- *Exact* mutants: Two mutants have the same binary string with the same erroneous output variables, and erroneous output values are exactly the same.

Table 10 shows distribution of these mutant relations, and their percentages out of total combinations (90525).

Relationship	Number of pairs	Percentage
Related mutants	1067	1.18%
Similar mutants	38	0.042%
Exact mutants	13	0.014%

Table 10 Mutants Relationship

4.5 Relationship between the Programs with Mutants

During the evaluation test, we also determined the correlation among the program version based on mutant executions. We defined two types of relationships: program versions with similar mutants, and program versions with exact mutants. The former includes program versions which generate similar mutants, while the latter includes those generating exact mutants. The results are shown, respectively, in Table 11 and Table 12. Each axis in these tables shows the program ID, and the values in the content, if any, indicate the number of similar or exact mutants between two corresponding program versions. Note these tables are symmetric.

Table 13 summarizes total program version pairs with similar and exact mutants. The pairs with exact mutants are interesting and valuable for analysis in detail. There are seven pairs of exact mutants. All these pairs were due to five exact faults, in which four exact fault occurs in two versions while one exact fault span three versions. Table 14 (a)-(e) provide a summary of these faults.

Here are the descriptions on the causes of these faults:

Pair 1 – Versions 4 and 8

The display mode is incorrectly calculated for a missing operation.

Pair 2 – Versions 12 and 31

Wrong calibration was made due to incorrect alignment access of array elements.

ID	01	02	03	04	05	07	08	09	12	15	17	18	20	22	24	26	27	29	31	32	33
01																					
02				02						02											
03																					
04		02					01			02	01	01					01				
05																					
07							02			02	01						01				
08				01		02				04	02	01									
09																					
12											01								01		
15		02		02		02	04				03										01
17				01		01	02		01	03											
18				01			01														
20																					
22																					
24																					
26																					
27				01		01															
29																					
31									01											01	
32																			01		
33										01											

Table 11 Program Versions with Similar Mutants

ID	01	02	03	04	05	07	08	09	12	15	17	18	20	22	24	26	27	29	31	32	33
01																					
02																					
03																					
04							01			01	01										
05																					
07																					
08				01																	
09																					
12																			01		
15				01							01										01
17				01						01											
18																					
20																					
22																					
24																					
26																					
27																					
29																					
31									01											01	
32																			01		
33										01											

Table 12 Program Versions with Exact Mutants

Pair 3 – Versions 15 and 33

Version 15 missed code to perform mod 4096 in calculating the average value in calibration. Version 33 missed code to ignore redundant data for calibration.

Pairs 4, 5, and 6 – Versions 4, 15, and 17

In estimation, all versions missed code to multiply a factor in calculation.

Relationship	Number of pairs	Percentage
Programs with Similar Mutants	19	9.05%
Programs with Exact Mutants	7	3.33%

Table 13 Summary of Program Relationship

	Version 4	Version 8
Module	Display Processor	Display Processor
Stage	Initcode	Initcode
Defect Type	Assign/Init	Assign/Init
Severity	C	C
Qualifier	Missing	Missing

Table 14 (a) Exact Pair 1: Versions 4 and 8

	Version 12	Version 31
Module	Calibrate	Calibrate
Stage	Initcode	Initcode
Defect Type	Algorithm/Method	Algorithm/Method
Severity	B	B
Qualifier	Incorrect	Incorrect

Table 14 (b) Exact Fault Pair 2: Versions 12 and 31

	Version 15	Version 33
Module	Calibrate	Calibrate
Stage	Initcode	Initcode
Defect Type	Algorithm/Method	Algorithm/Method
Severity	B	B
Qualifier	Missing	Missing

Table 14 (c) Exact Fault Pair 3: Versions 15 and 33

	Version 4	Version 15	Version 17
Module	Estimate Vehicle State	Estimate Vehicle State	Estimate Vehicle State
Stage	Initcode	Initcode	Initcode
Defect	Assign/Init	Assign/Init	Algorithm/Met

Type			hod
Severity	B	B	B
Qualifier	Incorrect	Incorrect	Incorrect

Table 14 (d) Exact Fault Pairs 4, 5, and 6: Versions 4, 15 and 17

	Version 31	Version 32
Module	Calibrate	Calibrate
Stage	Unit Test	Acceptance Test
Defect Type	Checking	Checking
Severity	B	B
Qualifier	Incorrect	Incorrect

Table 14 (e) Exact Fault Pair 7: Versions 31 and 32

Pair 7 – Versions 31 and 32

Version 31 contained an error in checking when checkout the sensors with excessive noise. Version 32 committed the same error in marking sensor status. These exact faults, however, were detected in different testing stages

We note that the amount of exact faults among program versions is very limited. This implies that design diversity involving multiple program versions can be an effective mechanism for software reliability engineering.

4.6 Major Findings in Dynamic Analysis

The major findings in the evaluation test are regarding the effectiveness of test coverage criteria and software design diversity. The coverage experiments show in average 61.5% faults can be detected with an increase of coverage. Therefore, looking for coverage increase is an effective means to detecting more faults. However, coverage measure itself does not guarantee such an indicator. High coverage of a test case does not necessarily lead to more fault detection.

The number of programs with exact mutants is very small, indicating the potential benefit of software fault tolerance. On the other hand, the number of related mutants is not negligible. Thus effective error detection and recovery schemes play a crucial role in distinguishing faults failing on the same data but with different results.

5. Software Testing using Domain Analysis

Zhao [31] proposed a new approach to generate test cases based on domain analysis of specifications and programs. In her technique, the differences of the functional domain and the operational domain are examined by analyzing the set of boundary conditions. Test cases are then designed by verifying the overlaps of operational domain and functional domain to locate the faults resulting from the discrepancies between these two domains.

Based on the new domain analysis approach we developed 90 new test cases, which are listed in Table 15. The major design principle of these test cases is to allow for exercising different legitimate boundaries of the operational domain not clearly identified in the specifications.

Case ID	Description
1-6	Modify *linStd* to short int boundary
7-16	Set *LinFailIn* array to short int boundary
17-25, 27-41, 42-65	Set *RawLin* to boundary
26,66, 67-73, 86	Modify *offRaw* array to boundary
74-79	Set *DisplayMode* in [–1..100] boundaries
80-85	Set *nsigTolerance* to various values
87-90	Set *base*=0, 99.999999, 999999, 1.000000, respectively

Note: Italic names in the table represent input variables

Table 15 Test Cases Generated by Domain Analysis

We executed these 90 test cases on the mutants we created. All 426 mutants can be killed by this test set. The diagram for individual test case contribution is depicted in Figure 6.

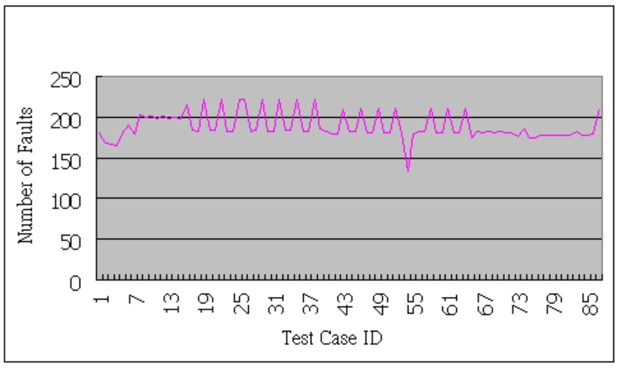

Figure 6 Contribution of Test Cases Generated by Domain Analysis

The average number of fault detected by each new test case, illustrated in Figure 6, is 183, ranging from 139 to 223. After redundant test cases are eliminated, a non-redundant set of 42 test cases is obtained, as shown in the black lines in Figure 7.

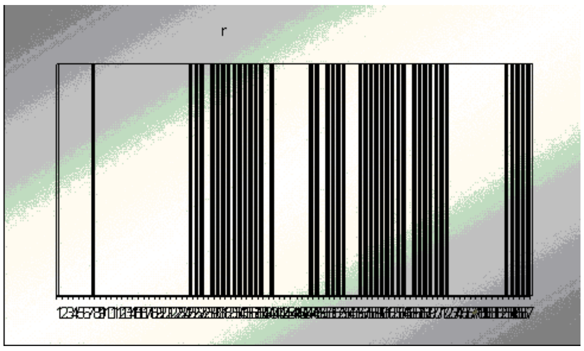

Figure 7 Non-redundant Test Set for Test Cases Generated by the Domain Analysis.

The newly designed test cases based on domain analysis display a completely different nature from that of the original test cases. In comparison with the non-redundant set of 698 test cases obtained from the original test set, this non-redundant set of only 42 test cases generated by the domain analysis approach is surprisingly effective. Both test sets kill all the 426 mutants, but the new set requires much less test cases. It is noted, however, that although both data sets (including 698 and 42 test cases, respectively) are non-redundant, they are not necessarily minimal test sets in killing all the 426 mutants

Furthermore, within the original 1200 test cases, each case in average can kill 248 mutants. For the newly generated 90 test cases, on the other hand, each test case can only kill an average of 183 mutants. When collected together in a test set, however, the new test cases can kill the same number of mutants more effectively. This implies that the capability of individual test case in killing mutants does not represent its capability in forming a minimal test set for an overall mutant coverage. A more critical factor is whether different test cases can explore different features of the program versions, thus killing different types of mutants.

Domain analysis helps generate test cases satisfying this critical factor. From our result, test cases generated based on boundary conditions via domain analysis are more effective in covering different aspects of the code in dealing with various border line cases within the operational domain. As this avoids producing test cases with similar capacities, the total number of test cases needed to detect the entire known fault set would tend to be smaller.

6. Conclusions

In this research effort we performed an empirical investigation on evaluating fault removal and fault tolerance issues as software reliability engineering techniques. We conducted a major experiment engaging multiple programming teams to develop a critical application whose specifications and test cases were obtained from the avionics industry. We applied mutation testing techniques with actual faults committed by programmers, and studied various aspects of the faults, including their nature, their manifestation process, their detectability, and their correlation. The evaluation results provided very positive support to current fault removal and fault tolerance techniques, with quantitative evidences.

Regarding the fault removal techniques, our experiment indicated that faults could be detected and removed with increase of testing coverage. We also observed some caveats about testing and fault tolerance: coverage measures and mutation scores cannot be evaluated in isolation, and an effective mechanism to distinguish related faults is critical.

We also conceived that a good test case should be characterized not only by its ability to detect more faults, but also by its ability to detect faults which are not detected by other test cases in the same test set. Our empirical data provided numerical supports to confirm this intuition. In our experiment, domain analysis was shown to be an effective approach to generating test cases. The newly generated test cases by this approach further revealed additional evidences that the individual fault detection capability of each test case in a test set does not represent the overall capability of the test set to cover more faults. Diversity natures of the test cases are more important.

Furthermore, our results implied that design diversity involving multiple program versions can be an effective solution for software reliability engineering, since the portion of program versions with exact faults is very small. The quantitative tradeoff between these two approaches, however, remains a research issue. Currently we can only generally perceive that software fault removal and software fault tolerance are complementary rather than competitive. As our future work, we will apply software reliability models on the program versions with similar and exact mutants to investigate their reliability, fault detection, and fault tolerance features.

References

[1] M. Lyu (ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, 1996.

[2] S. Rapps, E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information,” *IEEE Transactions on Software Engineering*, vol. SE-11, No.4, April 1985.

[3] W. E. Howden, “Weak Mutation Testing and Completeness of Test Sets,” *IEEE Transactions on Software Engineering*, vol. SE8, no. 4, July 1982, pp. 371-379.

[4] P.G. Frankl, E.J. Weyuker, “An Applicable Family of Data Flow Testing Criteria,” *IEEE Transactions on Software Engineering*, vol. SE-14, No. 10, October 1988.

[5] E.J.Weyuker, “The Cost of Data Flow Testing: An Empirical Study,” *IEEE Transactions on Software Engineering*, v.16 n.2, February 1990, pp.121-128.

[6] J.R. Horgan, S. London, and M.R. Lyu, “Achieving Software Quality with Testing Coverage Measures,” *IEEE Computer*, vol. 27, no. 9, September 1994, pp. 60-69.

[7] L. Briand and D. Pfahl, “Using simulation for assessing the real impact of test coverage on defect coverage,” *Proceedings of 10th IEEE International Symposium on. Software Reliability Engineering,* Nov. 1999, pp. 124-157.

[8] M.Chen, M.R. Lyu, and E. Wong, “Effect of Code Coverage on Software Reliability Measurement,” *IEEE Transactions on Reliability*, vol. 50, no. 2, June 2001, pp. 165-170.

[9] Y.K. Malaiya, N. Li, J. M. Bieman, and R. Karcich, "Software Reliability Growth with Test Coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, December 2002, pp. 420-426.

[10] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators,” *ACM Transactions on Software Engineering Methodology,* vol. 5, no. 2, 1996, pp. 99-118.

[11] A.J. Offutt, G. Rothermel, and C. Zapf, “An Experimental Evaluation of Selective Mutation,” *Proceedings of the 15th International Software Engineering Conference,* May 1993, pp. 1-107.

[12] W.E. Wong and A.P. Mathur, “Reducing the Cost of Mutation Testing: An Empirical Study,” *The Journal of Systems and Software*, vol. 31, no. 3, December 1995, pp. 185-196.

[13] A. Offutt and S. Lee, “An Empirical Evaluation of Weak Mutation,” *IEEE Transactions on Software Engineering,* vol. 20, no. 5, May 1994.

[14] E. Delamaro, C. Maldonado and A.P. Mathur, “Interface Mutation: An Approach for Integration Testing,” *IEEE Transactions on Software Engineering,* Vol. 27, No. 3, March 2001, pp. 228-247.

[15] A .J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *The Journal of Software Testing, Verification, and Reliability*, Vol 7, No. 3, September 1997, pp. 165-192.

[16] B. Randell and J. Xu, “The Evolution of the Recovery Block Concept,” *Software Fault Tolerance*, M. R. Lyu (ed.), Wiley, Chichester, 1995, pp. 1-21.

[17] A.A. Avizienis, “The Methodology of N-Version Programming,” *Software Fault Tolerance*, M. R. Lyu (ed.), Wiley, Chichester, 1995, pp. 23-46.

[18] J.C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, “Architectural Issues in Software Fault Tolerance,” *Software Fault Tolerance*, M. R. Lyu (ed.), Wiley, Chichester, 1995, pp. 47-80.

[19] K.E. Grosspietsch, “Optimizing the Reliability of the Component-based N-version Approaches,” *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS 2002),* Fort Lauderdale, Florida, April 2002, pp. 138-145.

[20] K.E. Grosspietsch, A. Romanovsky, “An Evolutionary and Adaptive Approach for N-version Programming,” *Proceedings of 27th Euromicro Conference*, Warsaw, Poland, September 2001, pp. 182-189.

[21] M. R. Lyu and Y. He, “Improving the N-version programming process through the evolution of a design paradigm,” *IEEE Transactions on Reliability*, 42(2), June 1993, pp. 179-189.

[22] B. Littlewood and D.Miller, “Conceptual Modeling of Coincident Failures in Multiversion Software,” *IEEE Transactions on Software Engineering*, vol. 15, no. 12, December 1989, pp. 1596-1614.

[23] M. Ege, M.A Eyler, M.U. Karakas, “Reliability Analysis in N-version Programming with Dependent Failures,” *Proceedings of Euromicro 27th Conference*, 2001, pp. 174 –181.

[24] X. Teng and H. Pham, “A Software-Reliability Growth Model for N-Version Programming Systems,” *IEEE Transactions on Reliability*, vol. 51, issue 3, September 2002, pp. 311-321.

[25] F. Belli and P. Jedrzejowicz, “Fault-Tolerant Programs and Their Reliability,” *IEEE Transactions on Reliability*, vol. 29(2), 1990, pp. 184-192.

[26] B. Littlewood, P. Popov, and L. Strigini, "Modelling Software Design Diversity - a Review," *ACM Computing Surveys*, Vol. 33, No. 2, June 2001, pp. 177-208.

[27] B. Littlewood, P. Popov, P. and L. Strigini, "Design Diversity: an Update from Research on Reliability Modelling," *Proceedings of Safety-Critical Systems Symposium 21, Bristol, U.K., Springer,* 2001.

[28] D.E. Eckardt and L.D. Lee, “A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors,” *IEEE Transaction on Software Engineering*, vol, SE-11, no. 12, December 1985, pp. 1511-1517.

[29] B. Littlewood, P. Popov and L. Strigini, "Assessing the Reliability of Diverse Fault-Tolerant Software-based Systems," *Safety Science*, vol40, 2002, pp.781-796.

[30] K. Kim, M. A. Vouk and D.F. McAllister, "An Empirical Evaluation of Maximum Likelihood Voting in High Inter- Version Failure Correlation Conditions," *Proceedings of 7th International Symposium on Software Reliability Engineering*, October 1996, pp 330-339.

[31] R. Zhao, *Research on Software Testing Methodologies, Ph.D. thesis*, Chinese Academy of Science, 2001.

[32] L.J. White and E.I. Cohen, "A Domain Testing Strategy," IEEE Transactions on Sofiware Engineering”, Vol. SE-6, No. 3, May 1980, pp. 247-257.

[33] D.E.Eckhardt, Caglavan, Knight, Lee, McAllister, Vouk, Kelly, “An experimental evaluation of software redundancy as a strategy for improving reliability,” *IEEE Transaction of Software Engineering*, vol 18, July 1991, pp 692-702.

[34] M.R. Lyu, J.R. Horgan, S. London, “A Coverage Analysis Tool for the Effectiveness of Software Testing,” *Proceedings of ISSRE’93*, Denver, November 1993, pp. 25-34.

[35] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong, “Orthogonal Defect Classification – A Concept for In-Process Measurements,” *IEEE Transactions on Software Engineering*, vol. 18, no. 19, November 1992, pp.943-956.

[36] Y. K. Malaiya, L. Naixin, J. Bieman, R. Karcich and B. Skibbe, “The Relationship Between Test Coverage and Reliability,” Proceedings of 5th International Symposium on Software Reliability Engineering, November 1994, pp. 186 -195.