The Chinese University of Hong Kong
Department of Computer Science and Engineering

Ph.D. – Term Paper

Title:	Testing Effectiveness and Fault Correlation Modeling for Diverse Software Systems		
Name:	CAI, Xia		
Student I.D.:	03440020		
Contact Tel. No.:	3163-4257	Email A/C:	xcai@cse.cuhk.edu.hk
Supervisor:	Prof. Michael R. Lyu		
Markers:	Prof. Ada Fu & Prof. Jeffrey Yu (SEEM)		
Mode of Study:	Full-time		
Submission Date:	April 20, 2005		
Term:	4		
Fields:			
Presentation Date:	April 28, 2005(Thursday)		
Time:	9:45–10:15 am		
Venue:	Rm. 1021, Ho Sin-Hang Engineering Building		

Testing Effectiveness and Fault Correlation Modeling for Diverse Software Systems

Abstract

While fault tolerant software is seen as a necessity, it is also controversial and a factor of regulatory uncertainty. Up to date researchers do not know what creditable reliability models for fault tolerant software are, how to test for fault tolerance, and how effective fault tolerant software can achieve. In particular, we cannot systematically develop models to predict reliability of fault tolerant software systems, and provide evidences regarding the validity of these models. One difficulty lies on the fact that there is no proper model to describe the nature and interactions of software faults regarding how they are manifested and how they are correlated. As currently, there are no quantitative assessment schemes for a comprehensive evaluation of fault tolerant software including model comparisons and trade-off studies with software testing techniques. This research is aimed at providing such an assessment for a systematic evaluation of fault tolerant software techniques.

In this term paper, we first survey the background, techniques, reliability modeling and applications for software fault tolerance. Then based on our previous experiment on software reliability analysis on fault tolerance, we conduct further experiments and analyze the experimental data to learn the correlations among these faults and the relation to their resulting failures. we apply the experimental data on current famous reliability modeling to examine their effectiveness.

Furthermore, we investigate the effectiveness of data flow coverage and mutation coverage in testing for design diversity. We examine different hypotheses on software testing and fault tolerance schemes, and find that code coverage is a positive indicator for fault detection capability of a given test set. Particularly, for exceptional test case, code coverage is clearly a strong indicator for its testing effectiveness.

Contents

Chapter 1

Introduction

Software permeates our modern society, and its complexity and criticality is ever increasing. There is an urgent need for a systematic development of highly reliable, continuously available, and extremely safe software. As faults are inevitable to software systems, fault tolerance is the system survival attribute which allows seamless delivery of expected service even after faults have manifested themselves within a software-intensive system.

While fault tolerant software is seen as a necessity, it is also controversial and a factor of regulatory uncertainty. Up to date researchers do not know what creditable reliability models for fault tolerant software are, how to test for fault tolerance, and how effective fault tolerant software can achieve. In particular, we cannot systematically develop models to predict reliability of fault tolerant software systems, and provide evidences regarding the validity of these models. One difficulty lies on the fact that there is no proper model to describe the nature and interactions of software faults regarding how they are manifested and how they are correlated. Several models have been proposed, yet debates among experts are frequent and heated. Moreover, there is lacking of real world project data for investigation on software testing and fault tolerance techniques together, with comprehensive analysis and evaluation. Without new research, it is doubtful that this impasse can be broken.

1.1 Fault Correlation Models for Design Diversity

Design diversity is one of the main techniques for software fault tolerance. This approach was proposed to achieve quality and reliability of software systems by detecting and tolerating software faults during operation. Its basic idea is to employ different development teams in building different program versions independently according to one single specification [58]. During program executions, the final consensus output is either voted by multiple versions, or verified by an acceptance test, which can be one of the program versions. The multi-version programs are expected to fail with low probability of coincident failures. Although many research efforts have *been conducted for investigation, experimentation, modeling and evaluation of software design diversity, it still remains a debatable approach compared with other software engineering techniques. One main reason is the lack of real world project data on collecting the features of design diversity; and the other is the failures in diverse versions may not occur independently, making it difficult to establish justifiable predictive reliability models.*

Nevertheless, to attempt the modeling of reliability and fault correlations achieved in design diversity, some methods have been proposed. Eckhardt and Lee [25] proposed the first model of fault correlation for diverse systems. Later Littlewood and Miller [53] showed a conceptual model in which the reliability of a pair of versions may even be better than what is under the assumption of independence. Dugan and Lyu [23] proposed a dependability model for N-version programming to parameterize the possibility of fault correlations. Recently, Popov Strigini et al [73] further pointed out that the bounds on the reliability of multiple-version systems can be estimated by dividing the demand space of the test cases into disjoint sub-domains.

1.2 The Effectiveness of Software Testing

As the main fault removal technique, software testing is one of the most effort-intensive activities during software development [7]. The key issue in software testing is test case selection and evaluation. An effective test set should detect software faults that do not easily lead to failure by other test cases. To improve the test resource allocation, code coverage has been proposed as an indicator of testing effectiveness and completeness for the purpose of test case selection and evaluation [63, 78, 81]. Code coverage is measured as the fraction of program codes that are executed at least once during the test. Various code coverage criteria have been suggested [36], including block coverage, decision coverage, C-use coverage and P-use coverage, etc.

However, it remains a controversial issue about whether code coverage is a good indicator for fault detection capability of test cases. Some previous studies show that high code coverage brings high software reliability and low fault rate [28, 36, 78, 89]. Such experimental data indicate that both code coverage and fault detected in programs grow over time, as testing progresses. For example, [16] observed this correlation between the code coverage and software reliability using experimentation with randomly generated flow graphs. In [92], it is reported that the correlation between test effectiveness and block coverage is higher than that between test effectiveness and the size of test set. [29] showed that an increase in reliability comes with an increase in at least one code coverage measures, and a decrease in reliability is accompanied by a decrease in at least one code coverage measures.

Furthermore, considering code coverage is a positive indicator for software reliability and quality, some researchers try to model the relationship between code coverage and code quality by hypergeometric distribution *modeling [90] (under the assumption of a uniform probability and a random distribution of defects in the unit code, and independence between defects). Some suggest code coverage as an additional parameter for the prediction of software failures in operation [15]. Some model the relation among testing time, coverage and reliability altogether [65].*

On the other hand, despite the observations of correlation existing in code coverage and fault coverage, a question is raised [14]: Can this phenomenon of concurrent growth be attributed to a causal dependency between code coverage and fault detection, or is it just coincidental due to the cumulative nature of both measures? A simulation experiment involving Monte Carlo simulation was conducted on the assumption that there is no causal dependency between code coverage and fault detection. The testing result on published data did not support a causal dependency between code coverage and defect coverage.

Overall, the relationship between code coverage and fault detection is very complicated. More empirical data and theoretical insight are needed to explore the causal dependency between the two measures. In our previous work, we performed mutation testing and code coverage testing [64]. The results indicate that in most situations additional coverage of the code was achieved when the mutants were killed by a new test case. It observes that the increase in code coverage is related to more fault detections by a large portion (61.5%) among 21 program versions, although the range (from 22.2% to 94.7%) is very wide among different versions. In this paper, we will further study the relationship between code coverage and fault detection capability under different testing profiles.

1.3 Organization of this Term Paper

As described in the term paper of last year, in order to obtain new real-world data regarding fault tolerant software. we conducted fault tolerant software experiments which applying coverage-based and mutation-based testing techniques, and collect data for detailed analysis. Comprehensive experimentation was performed to study the nature, source, type, detectability, and effect of faults uncovered in the program versions.

What have been done in the past year are listed as follows:

- *Survey the background, techniques, reliability modeling and applications for software fault tolerance [1];*
- *Conduct further experiments and analyze the experimental data to learn the correlations among these faults and the relation to their resulting failures. Apply our experimental data on current famous reliability modeling to examine their effectiveness [2].*

[1]This is an invited paper for Encyclopedia of Computer Science and Engineering, to be published by Wiley

[2]Xia Cai and Michael R. Lyu, ”An Empirical Study on Reliability and Fault Correlation Models for Diverse Software Systems”, ISSRE’2003, St-Malo,France, Nov.2003

- *Investigate the effectiveness of data flow coverage and mutation coverage in testing for design diversity. We examine different hypotheses on software testing and fault tolerance schemes, and find that code coverage is a positive indicator for fault detection capability in software testing with our software fault tolerance project* [3].

The remainder of this term paper is organized as follows. The review of software fault tolerance and software reliability modeling are listed in the Chapter 2 (Background Stduy). The experimental setup and procedures are also mentioned in this chapter. Then we conduct further experiments and analysis on the correlations among faults in design diversity and compare the performance of current reliability models, which are included in Chapter 3. Chapter 4 shows our new findings about the effect of code coverage on fault detection capability in fault-tolerance software testing. Finally, Chapter 5 concludes this term paper by stating our conclusion and future work.

[3] Xia Cai and Michael R. Lyu, "The Effect of Code Coverage on Fault Detection Capability under Different Testing Profiles", A-MOST of ICSE'2005, St. Louis, Missouri, May, 2005

Chapter 2

Background Study

Fault tolerance is the survival attribute of a system or component to continue operating as required despite the manifestation of hardware or software faults [39]. Fault-tolerant software is concerned with all the techniques necessary to enable a software system to tolerate software design faults remaining in the system after its development [58]. When a fault occurs, fault-tolerant software provides mechanisms to prevent the system failure from occurring [75].

Fault-tolerant software delivers continuous service complying with the relevant specification in the presence of faults typically by employing either single version software techniques or multiple version software techniques. We will address four key perspectives for fault-tolerant software: historical background, techniques, modeling schemes and applications.

2.1 Historical Background

Most of the fault-tolerant software techniques were introduced or brought up in 1970s. For example, as one of single version fault-tolerant software techniques, early research and discussions on exception handling began to appear in the 1970s and led to more mature definitions, terminology and exception mechanisms later on [19]. Another technique, checkpointing and recovery, was also commonly used to enhance software reliability and increase their computational efficiency, as shown in the literatures [68].

In the early 1970s, a research project was conducted at the University of Newcastle [77]. The idea of the recovery block (RB) evolved from this project and it became one of the methods currently used for safety-critical software. Recovery block is one of three main approaches in so-called design diversity, which is also known as multi-version fault-tolerant software techniques. N-version programming was introduced in 1977 [5], which involved redundancy of three basic elements in the approach: process, product and environment [4]. N self-

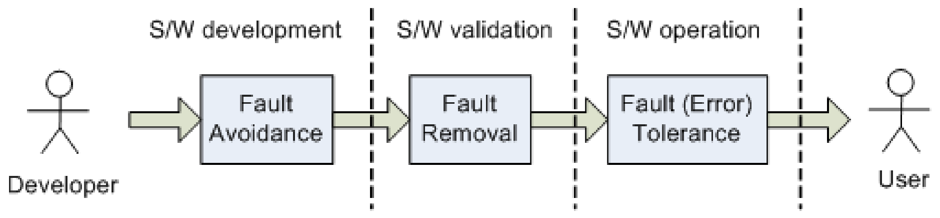

Figure 2.1. The Transition of Fault, Error and Failure

checking programming approach was introduced long after the previous two methods, yet it based on the concept of self-checking programming which has long been introduced [49].

Since then, many other approaches and techniques have been proposed for fault-tolerant software, and various models and experiments have been employed to investigate the features of these approaches. Some of them will be addressed in the following part of this paper.

2.1.1 Definitions

As fault-tolerant software is capable of providing the expected service despite the presence of software faults [5, 76], we first introduce the concepts related to this technique [51].

Failures. A failure occurs when the user perceives that a software program is unable to deliver the expected service [49]. The expected service is described by a system specification or a set of user requirements.

Errors. An error is part of the system state which is liable to lead to a failure. It is an intermediate stage in between faults and failures. An error may propagate, i.e., produce other errors.

Faults. A fault, sometimes called a bug, is the identified or hypothesized cause of the software failure. Software faults can be classified as design faults and operational faults according to the phases of creation. Although the same classification can be used in hardware faults, we only interpret them in the sense of software here.

Design faults. A design fault is a fault occurring in software design and development process. Design faults can be recovered with fault removal approaches by revising the design documentation and the source code.

Operational faults. An operational fault is a fault occurring in software operation due to timing, race conditions, workload-related stress and other environmental conditions. Such a fault can be removed by recovery, i.e., rollback to the initial state and executed again.

Fault-tolerant software thus attempts to prevent failures by tolerating software errors caused by design faults. The progression "fault-error-failure" shows their causal relationship, as shown in Figure 2.1. There are two major groups of approaches to deal with design faults: 1) fault avoidance (prevention) and fault removal during the software development process, and 2) fault tolerance and fault/failure forecasting after the development process.

These terms can be defined as follows:

Fault avoidance (prevention). To avoid or prevent the introduction of faults by engaging various design methodologies, techniques and technologies, including structured programming, object-oriented programming, software reuse, design patterns and formal methods.

Fault removal. To detect and eliminate software faults by techniques such as reviews, inspection, testing, verification and validation.

Fault tolerance. To provide a service complying with the specification in spite of faults, typically by means of single version software techniques or multi-version software techniques. Note that, although fault tolerance is a design technique, it handles manifested software faults during software operations. Although software fault tolerance techniques are proposed to tolerant software errors, they can help to tolerate hardware faults as well.

Fault/failure forecasting. To estimate the existence of faults and the occurrences and consequences of failures by dependability-enhancing techniques consisting of reliability estimation and reliability prediction.

2.1.2 Rationale

The principle of fault-tolerant software is to deal with residual design faults. For software systems, the major cause of residual design faults can be complexity, difficulty and completeness involved in software design, implementation and testing phases. The aim of fault-tolerant software, thus, is to prevent software faults from incorrect operations, including severe situations such as hanging or as the worst, crashing a system. To achieve this purpose, appropriate structuring techniques should be applied for proper error detection and recovery. Nevertheless, fault tolerance strategies should be simple, coherent and general in their application to all software systems. Moreover, they should be capable of coping with multiple errors, including the ones detected during the error recovery process itself, which is usually deemed fault-prone due to its complexity and lack of thorough testing.

To satisfy these principles, strategies like checkpointing, exception handling and data diversity are designed for single version software, while recovery block (RB), N-version programming (NVP) and N self-checking programming (NSCP) have been proposed for multi-version software. The details of these techniques and their strategies are discussed in Section 3.

2.1.3 Practice

From a user's point of view, fault tolerance represents two dimensions: availability and data consistency of the application [38]. Generally, there are four layers of fault tolerance. The top layer is composed of general fault tolerance techniques which are applicable in all applications, including checkpointing, exception handling, recovery

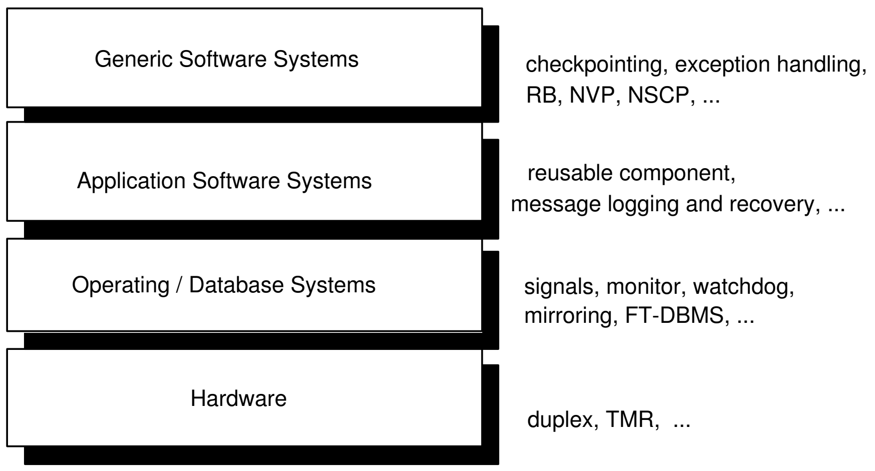
Generic Software Systems
checkpointing, exception handling, RB, NVP, NSCP, ...
Application Software Systems
reusable component, message logging and recovery, ...
Operating / Database Systems
signals, monitor, watchdog, mirroring, FT-DBMS, ...
Hardware
duplex, TMR, ...

Figure 2.2. Layers of Fault Tolerance

block, N-version programming, N-self checking programming and other approaches. Some of the top-level techniques will be addressed in the following section. The second layer consists of application-specific software fault tolerance techniques and approaches such as reusable component, fault-tolerant library, message logging and recovery, etc. The next layer involves the techniques deployed on the level of operating and database systems, e.g., signal, watchdog, mirroring, fault-tolerant database (FT-DBMS), transaction and group communications. Finally, the underlying hardware also provides fault-tolerant computing and network communication services for all the upper layers, i.e., duplex, triple modular redundancy (TMR), symmetric multiprocessing (SMP), shared memory and so on. Summary of different layers for fault tolerance techniques and approaches are shown in Figure 2.2.

Technologies and architectures have been proposed to provide fault tolerance for some mission-critical applications. These applications include airplane control systems (e.g., Boeing 777 airplane and AIRBUS A320/A330/A340 aircraft) [11, 35], aerospace applications [67], nuclear reactors, telecommunications products [38], network systems [46], and other critical software systems.

2.2 Fault-Tolerant Software Techniques

We examine two different groups of techniques for fault-tolerant software: single version and multi-version software techniques [58]. Single version techniques involve improving the fault detection and recovery features of a single piece of software on top of fault avoidance and removal techniques. The basic fault-tolerant features include program modularity, system closure, atomicity of actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity [58, 85].

In more advanced architectures, design diversity is employed where multiple software versions are developed

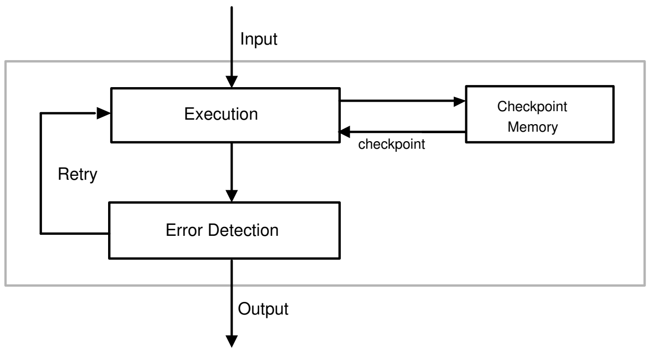

Figure 2.3. Logic of checkpoint and recovery

independently by different program teams using different design methods, yet they provide the equivalent service according to the same requirement specifications. The main techniques of this multiple version software approach are recovery blocks, N-version programming, N self-checking programming, and other variants based on these three fundamental techniques.

All the fault-tolerant software techniques can be engaged in any artifact of a software system: procedure, process, software program, or the whole system including the operating system. The techniques can also be selectively applied to those components especially prone to faults because of the design complexity.

2.2.1 Single Version Software Techniques

Single-version fault tolerance is based on temporal and spacial redundancies applied to a single version of software to detect and recover from faults. Single-version fault-tolerant software techniques include a number of approaches. We focus our discussions on two main methods: checkpointing and exception handling.

Checkpointing and Recovery

For single-version software, the technique most often mentioned is the checkpoint and recovery mechanism [74]. Checkpointing is used in (typically backward) error recovery, by saving the state of a system periodically. When a failure is detected, the previous state is recalled and the whole system is restored to that particular state. A recovery point is established when the system state is saved, and discarded if the process result is acceptable. The basic idea of checkpointing is shown in Figure 2.3. It has the advantages of being independent of the damage caused by a fault.

The information saved for each state includes the values of variables in the process, its environment, control information, register values, and so on. Checkpoints are snapshots of the state at various points during the

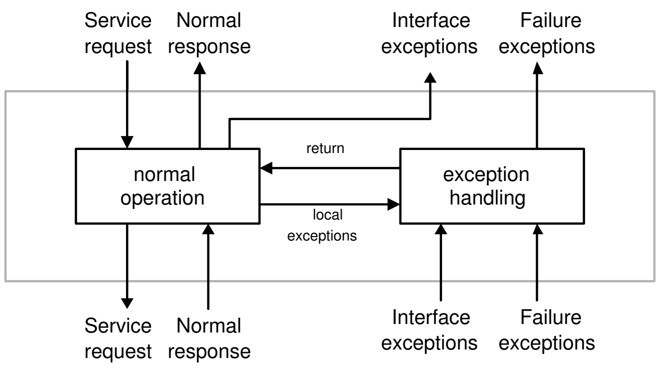

Figure 2.4. Logic of exception handling

execution.

There are two kinds of checkpointing and recovery schemes: single process systems with a single node, and multiple communicating processes on multiple nodes [75]. For single process recovery, a variety of different strategies is deployed to set the checkpoints. Some strategies use randomly-selected points, some maintain a specified time interval between checkpoints, and others set a checkpoint after a certain number of successful transactions have been completed.

For multiprocess recovery, there are two approaches: asynchronous and synchronous checkpointing. The difference between the two is that the checkpointing by the various nodes in the system is coordinated in synchronous checkpointing, but not coordinated in asynchronous checkpointing. Different protocols for state saving and restoration have been proposed for the two approaches [75].

Exception Handling

Ideal fault-tolerant software systems should recognize interactions of a component with its environment, provide a means of system structuring that make it easy to identify what part of the system to use to cope with each kind of error, and provide normal and abnormal (i.e., exception) responses within a component and among components' interfaces [52]. The structure of exception handling is shown in Figure 2.4.

Exception handling, proposed in the 1970's [31], is often considered as a limited approach to fault-tolerant software [18]. Since departure from specification is likely to occur, exception handling aims at handling abnormal responses by interrupting normal operations during program execution. In fault-tolerant software, exceptions are signaled by the error detection mechanisms as a request for initiation of an appropriate recovery procedure. The design of exception handlers requires consideration of possible events that can trigger the exceptions, prediction of the effects of those events on the system, and selection of appropriate mitigating actions.

A component generally needs to cope with three kinds of exceptional situations: interface exceptions, local exceptions and failure exceptions. Interface exceptions are signaled when a component detects an invalid service request. This type of exception is triggered by the self-protection mechanisms of the component and is treated by the component that made the invalid request. Local exceptions occur when a component's error detection mechanisms find an error in its own internal operations. The component returns to normal operations after exception handling. Failure exceptions are identified by a component after it has detected an error that its fault processing mechanisms were unable to handle successfully. In effect, failure exceptions notify the component making the service request that it has been unable to provide the requested service.

2.2.2 Multi-version Software Techniques

The multi-version fault-tolerant software technique is the so-called design diversity approach. This involves developing two or more versions of a piece of software according to the same requirement specifications. The rationale for the use of multiple versions is the expectation that components built differently (i.e., different designers, different algorithms, different design tools, etc) should fail differently [5]. Therefore, in the case that one version fails in a particular situation, there is a good chance that at least one of the alternate versions is able to provide an appropriate output.

These multiple versions are executed either in sequence or in parallel, and can be used as alternatives (with separate means of error detection), in pairs (to implement detection by replication checks) or in larger groups (to enable masking through voting). Three fundamental techniques are known as recovery block, N-version programming and N self-checking programming.

Recovery Block

The recovery block technique involves multiple software versions implemented differently such that an alternative version is engaged after an error is detected in the primary version [76, 77]. The question of whether there is an error in the software result is determined by an acceptance test (AT). Thus the recovery block uses an acceptance test and backward recovery to achieve fault tolerance. As the primary version will be executed successfully most of the time, the most efficient version is often chosen as the primary alternate and the less efficient versions are placed as secondary alternates. Consequently, the resulting rank of the versions reflects, in a way, their diminishing performance.

The usual syntax of the recovery block is as follows. First of all, the primary alternate is executed; if the output of the primary alternate fails the acceptance test, a backward error recovery is invoked to restore the previous

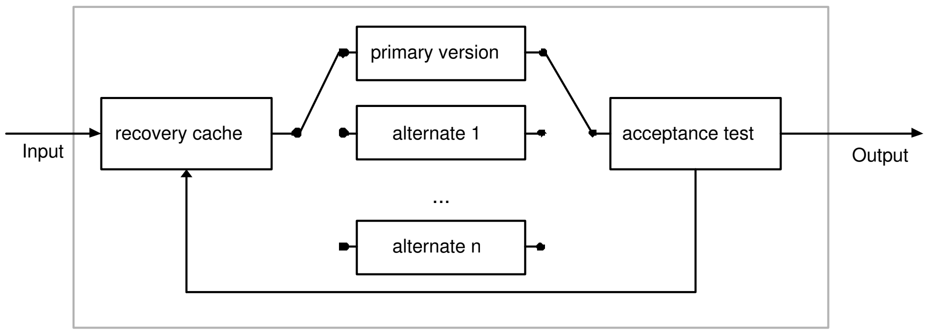

Figure 2.5. The recovery block (RB) model

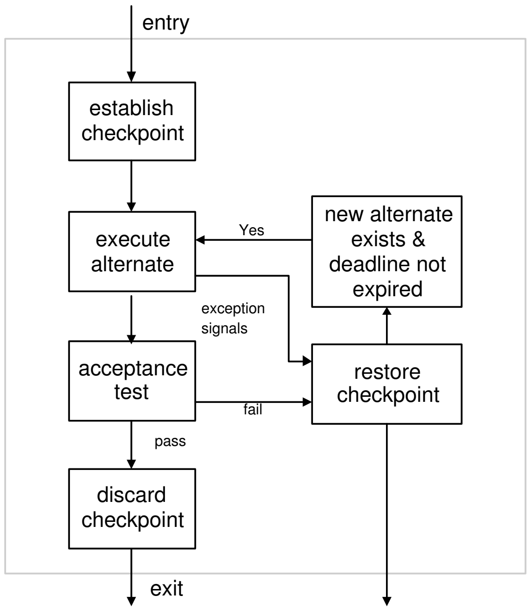

Figure 2.6. Operation of recovery block

state of the system, then the second alternate will be activated to produce the output; similarly, every time an alternate fails the acceptance test, the previous system state will be restored and a new alternate will be activated. Therefore, the system will report failure only when all the alternates fail the acceptance test, which may happen with a much lower probability than in the single version situation. The recovery block model is shown in Figure 2.5, while the operation of the recovery block is shown in Figure 2.6.

The execution of the multiple versions is usually sequential. If all the alternate versions fail in the acceptance test, the module must raise an exception to inform the rest of the system about its failure.

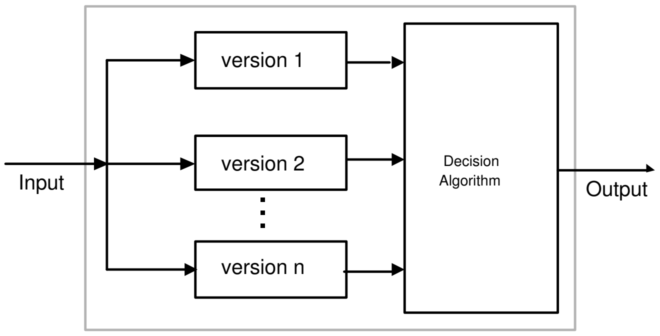

Figure 2.7. The N-version programming (NVP) model

N-Version Programming

The concept of N-version programming (NVP) was first introduced in 1977 [5]. It is a multi-version technique in which all the versions are typically executed in parallel and the consensus output is based on the comparison of the outputs of all the versions [58]. In the event that the program versions are executed sequentially due to lack of resources, it may require the use of checkpoints to reload the state before a subsequent version is executed. The N-version software model is shown in Figure 2.7.

The NVP technique uses a decision algorithm (DA) and forward recovery to achieve fault tolerance. The use of a generic decision algorithm (usually a voter) is the fundamental difference of NVP from the RB approach, which requires an application-dependent acceptance test. The complexity of the decision algorithm is generally lower than that of the acceptance test. In NVP, since all the versions are built to satisfy the same specification, it requires considerable development effort but the complexity (i.e., development difficulty) is not necessarily much greater than that of building a single version. Much research has been devoted to the development of methodologies that increase the likelihood of achieving effective diversity in the final product [4, 9, 24, 47].

N-Self Checking Programming

N self-checking programming (NSCP) was developed in 1987 by Laprie et al. [50, 49]. It involves the use of multiple software versions combined with structural variations of the recovery block and N-version programming approaches. Both acceptance test and decision algorithms can be employed in NSCP to validate the outputs of multiple versions.

The N self-checking programming method employing acceptance tests is shown in Figure 2.8. Same as RB and NVP, the versions and the acceptance tests are developed independently but each designed to fulfill the requirements. The main difference of NSCP from the RB approach is in its use of different acceptance tests for different

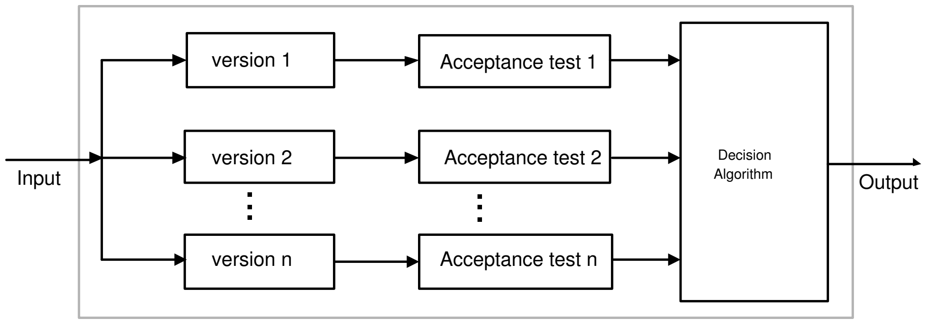

Figure 2.8. N self-checking programming using acceptance test

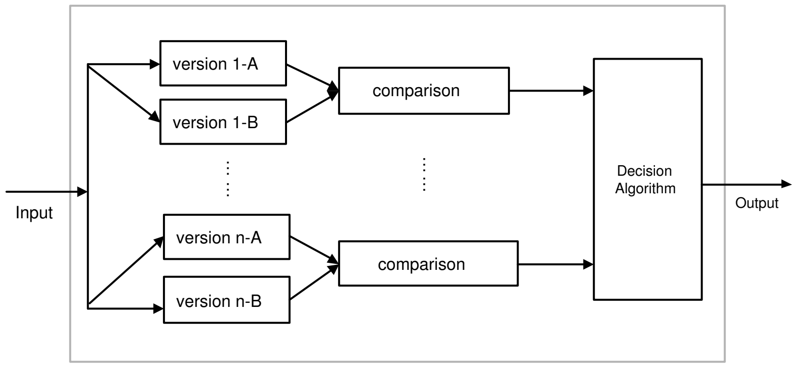

Figure 2.9. N self-checking programming using decision algorithm

versions. The execution of the versions and tests can be done sequentially or in parallel but the output is taken from the highest-ranking version that passes its acceptance test. Sequential execution requires a set of checkpoints, and parallel execution requires input and state consistency algorithms.

N self-checking programming engaging decision algorithms for error detection is shown in Figure 2.9. Similar to N-version programming, this model has the advantage of using an application-independent decision algorithm to select a correct output. This variation of self-checking programming has the theoretical vulnerability of encountering situations where multiple pairs pass their comparisons but the outputs differ between pairs. That case must be considered and an appropriate decision policy should be selected during the design phase.

Comparison among RB, NVP and NSCP

Each design diversity technique, recovery block, N-version programming, and N self-checking programming, has its own advantages and disadvantages compared with the others. We compare the features of the three and list them in Table 2.1.

The differences between acceptance test (AT) and decision algorithm (DA) are: 1) AT is more complex and

Table 2.1. Comparison of design diversity techniques

Features	RB	NVP	NSCP
Minimum number of versions	2	3	4
Output mechanism	Acceptance Test	Decision Algorithm	Decision Algorithm and Acceptance Test
Execution time	primary version	slowest version	slowest pair
Recovery scheme	backward recovery	forward recovery	forward and backward recovery

difficult in implementation, but it can still produce correct output when multiple distinct solutions exist in multiple versions; 2) DA is more simple, efficient and liable to produce correct output since it is just a voting mechanism; but it is less able to deal with multiple solutions.

Other Techniques

Besides the three fundamental design diversity approaches listed above, there are some other techniques available, essentially variants of RB, NVP and NSCP. They include consensus recovery block, distributed recovery block, hierarchical N-version programming, t/(n-1)-variant programming, and others. Here we introduce some of these techniques briefly.

Distributed Recovery Block

The distributed recovery block (DRB) technique, developed by Kim in 1984 [45], is adopted in distributed and/or parallel computer systems to realize fault tolerance in both hardware and software. DRB combines recovery blocks and a forward recovery scheme to achieve fault tolerance in real-time applications. The DRB uses a pair of self-checking processing nodes (PSP) together with both the software-implemented internal audit function and the watchdog timer to facilitate real-time hardware fault tolerance. The basic DRB technique consists of a primary node and a shadow node, each cooperating with a recovery block, and the recovery blocks execute on both nodes concurrently.

Consensus Recovery Block

The consensus recovery block approach combines N-version programming and the recovery block technique to improve software reliability [79]. The rationale of consensus recovery blocks is that RB and NVP each may suffer from its specific faults. For example, the RB acceptance tests may be fault-prone, and the decision algorithm in NVP may not be appropriate in all situations, especially when multiple correct outputs are possible. The consensus recovery block approach employs a decision algorithm as the first layer decision. If a failure is detected in the first *layer, a second layer using acceptance tests is invoked. Obviously having more levels of checking than either RB or NVP, consensus recovery block is expected to have an improved reliability.*

t/(n-1)-Variant Programming

t/(n-1)-variant programming (VP) was proposed by Xu and Randell in 1997 [93]. The main feature of this approach lies in the mechanism engaged in selecting the output among the multiple versions. The design of the selection logic is based on the theory of system-level fault diagnosis. The selection mechanism of t/(n-1)-VP has a complexity of O(n) - less than some other techniques - and it can tolerate correlated faults in multiple versions.

2.3 Modeling Schemes on Design Diversity

There have been numerous investigations, analyses and evaluations of the performance of fault-tolerant software techniques in general and of the reliability of some specific techniques [75]. Here we list only the main modeling and analysis schemes that assess the general effectiveness of design diversity.

To evaluate and analyze both the reliability and the safety of various design diversity techniques, different modeling schemes have been proposed to capture design diversity features, describe the characteristics of fault correlation between diverse versions, and predict the reliability of the resulting systems. The following modeling schemes are discussed in chronological order.

2.3.1 Eckhardt and Lee's Model

Eckhardt and Lee (EL Model) [25] proposed the first probability model that attempts to capture the nature of failure dependency in N-version programming. The EL model is based on the notion of "variation of difficulty" over the user demand space. Different parts of the demand space present different degrees of difficulty, making the program versions built independently more likely to fail with the same "difficult" parts of the target problem. Therefore, failure independency between program versions may not be the necessary result of "independent" development when failure probability is averaged over all demands. For most situations, in fact, positive correlation between version failures may be exhibited for a randomly chosen pair of program versions.

2.3.2 Littlewood and Miller's Model

Littlewood and Miller [53] (LM model) showed that the variation of difficulty could be turned from a disadvantage into a benefit with forced design diversity [73]. "Forced" diversity may insist that different teams apply different development methods, different testing schemes, and different tools and languages. With forced diversity, a problem that is more difficult for one team may be easier for another team (and vice versa). The possibility of *negative correlation between two versions means that the reliability of a 1-out-of-2 system could be greater than it would be under the assumption of independence. Both EL and LM models are "conceptual" models because they do not support predictions for specific systems and they depend greatly on the notion of difficulty defined over the possible demand space.*

2.3.3 Dugan and Lyu's Dependability Model

The dependability model proposed by Dugan and Lyu in [23] provides a reliability and safety model for fault-tolerant hardware and software systems using a combination of fault tree analysis and the Markov modeling process. The reliability/safety model is constructed by three parts: a Markov model details the system structure, and two fault trees represent the causes of unacceptable results in the initial configuration and in the reconfigured state. Based on this three-level model, the probability of unrelated and related faults can be estimated according to experimental data.

In a reliability analysis study [23], the experimental data showed that DRB and NVP performed better than NSCP. In the safety analysis, NSCP performed better than DRB and NVP. In general, their comparison depends on the classification of the experimental data.

2.3.4 Tomek and Trivedi's Stochastic Reward Nets Model

Stochastic reward nets (SRNs) are a variant of stochastic Petri nets. SRNs are employed in [84] to model three types of fault-tolerant software systems: RB, NVP and NSCP. Each SRN model is incorporated with the complex dependencies associated with the system, such as correlation failures and separate failures, detected faults and undetected faults. A Markov reward model underlies the SRN model. Each SRN is automatically converted into a Markov reward model to obtain the relevant measures. The model has been parameterized by experimental data in order to describe the possibility of correlation faults.

2.3.5 Popov and Strigini's Reliability Bounds Model

Popov and Strigini attempted to bridge the gap between the conceptual models and the structural models by studying how the conceptual model of failure generation can be applied to a specific set of versions [73]. This model estimates the probability of failure on demand given the knowledge of subdomains in a 1-out-of-2 diverse system. Various alternative estimates are investigated for the probability of coincident failures on the whole demand space as well as in subdomains. Upper bounds and likely lower bounds for reliability are obtained by using data from individual diverse versions. The results show the effectiveness of the model in different situations *having either positive or negative correlations between version failures.*

2.3.6 Experiments and Evaluations

Experiments and evaluations are necessary to determine the effectiveness and performance of different fault-tolerant software techniques and the corresponding modeling schemes. Various projects have been conducted to investigate and evaluate the effectiveness of design diversity, including UCLA Six-Language project [42, 58], NASA 4-University project [24, 73, 88], Knight and Leveson's experiment [47], Lyu-He study [23, 61], etc.

These projects and experiments can be classified into three main categories: 1) evaluations on the effectiveness and cost issues of the final product of diverse systems [1, 5, 10, 41, 43, 47, 34]; 2) experiments evaluating the design process of diverse systems [4]; and 3) adoption of design diversity into different aspects of software engineering practice [61, 64].

To investigate the effectiveness of design diversity, an early experiment [5], consisting of running sets of student programs as 3-version fault-tolerant programs, demonstrated that the N-version programming scheme worked well with some sets of programs tested, but not others. The negative results were natural since inexperienced programmers cannot be expected to produce highly reliable programs. Another student-based experiment [47] involved 27 program versions developed differently. Test cases were conducted on these program versions in single and multiple version configurations. The results showed that N-version programming could improve reliability; yet correlated faults existed in various versions, adversely affecting design diversity. In another study, Kelly et al. [43] conducted a specification diversity project, using two different specifications with the same requirements. Anderson et al. [1] studied a medium-scale naval command and control computer system developed by professional programmers through the use of the recovery block. The results showed that 74% of the potential failures could be successfully masked. Another experiment evaluating the effectiveness of design diversity is the Project on Diverse Software (PODS) [10]. This consisted of three diverse teams implementing a simple nuclear reactor protection system application. There were two diverse specifications and two programming languages adopted in this project. With good quality control and experienced programmers, high quality programs and fault-tolerant software systems were achieved.

For the evaluation of the cost of design diversity, Hatton [34] collected evidence to indicate that diverse fault-tolerant software techniques are more reliable than producing one good version, and more cost effective in the long run. Kanoun [41] analyzed work hours spent on variant design in a real-world study. The results showed that costs were not doubled by developing a second variant.

In a follow-up to the work of Avizienis and Chen [5], a six language NVP project was conducted using a *proposed N-version Software Design Paradigm [57]. The NVP paradigm was composed of two categories of activities: standard software development procedures and concurrent implementation of fault tolerance techniques. The results verified the effectiveness of the design paradigm in improving the reliability of the final fault-tolerant software system.*

To model the fault correlation and measure the reliability of fault-tolerant software systems, experiments have been employed to validate different modeling schemes. The NASA 4-University project [88] involved 20 two-person programming teams. The final twenty programs went through a three-phase testing process, namely, a set of 75 test cases for acceptance test, 1100 designed and random test cases for certification test, and over 900,000 test cases for operational test. The same testing data have been widely employed [24, 53, 73] to validate the effectiveness of different modeling schemes.

The Lyu-He study [61] was derived from an experimental implementation involving 15 student teams guided by the evolving NVP design paradigm in [4]. Moreover, a comparison was made between the NASA 4-University project, the Knight-Leveson experiment, the Six-Language project and the Lyu-He experiment in order to further investigate and discuss the effectiveness of design diversity in improving software reliability. The results were further used in [23] to evaluate the prediction accuracy of Dugan and Lyu's Model. Lyu et al [64] reported a multi-version project on The Redundant Strapped-Down Inertial Measurement Unit (RSDIMU), the same specification employed in the NASA 4-University project. The experiment developed 34 program versions, from which 21 versions were selected to create mutants. Following a systematic rule for the mutant creation process, 426 mutants, each containing a real program fault identified during the testing phase, were generated for testing and evaluation. The testing results were subsequently engaged to investigate the probability of related and unrelated faults using the PS and DL models.

Current results indicate that for design diversity techniques, NSCP is the best candidate to produce a safe result, while DRB and NVP tend to achieve better reliability than NSCP, although the difference is not significant.

2.4 Applications

There are many application-level methodologies for fault-tolerant software techniques. As we have indicated, the applications include airplane control systems (e.g., Boeing 777 airplane [35] and AIRBUS A320/A330/A340 aircraft [87]), aerospace applications [67], nuclear reactors, telecommunications products [38], network systems [46], and other critical software systems such as wireless network, grid-computing, etc. Most of the applications adopt single version software techniques for fault tolerance, i.e., reusable component, checkpointing and recovery, etc. The design diversity approach has only been applied in some mission-critical applications, e.g., airplane *control systems, aerospace, and nuclear reactor applications. There are also emerging experimental investigations into the adoption of design diversity in practical software systems, such as SQL database servers [72].*

We may summarize the fault-tolerant software applications into four categories: 1) reusable component library, e.g., [38]; 2) checkpointing and recovery schemes, e.g., [17, 74]; 3) entity replication and redundancy, e.g., [40, 86]; 4) early applications and projects on design diversity, e.g., [35, 72, 87]. An overview of some of these applications is given below.

Huang and Kintala [38] developed three cost-effective reusable software components, i.e., watchd, libft, and REPL, to achieve fault tolerance in the application level based on availability and data consistency. These components have been applied to a number of telecommunication products.

According to [74], the new mobile wireless environment poses many challenges for fault-tolerant software due to the dynamics of node mobility and the limited bandwidth. Particular recovery schemes are adopted for the mobile environment. The recovery schemes combine a state-saving strategy and a handoff strategy, including two approaches (No Logging and Logging) for state-saving, and three approaches (Pessimistic, Lazy, and Trickle) for handoff. Chen and Lyu [17] have proposed a message logging and recovery protocol on top of the CORBA architecture. This employs the storage available at the access bridge to log messages and checkpoints of a mobile host in order to tolerate mobile host disconnection, mobile host crash and access bridge crash.

Entity replication and modular redundancy are also widely used in application software and middleware. Townend and Xu [86] proposed a fault-tolerant approach based on job replication for Grid computing. This approach combines a replication-based fault tolerance approach with both dynamic prioritization and dynamic scheduling. Kalbarczyk et al [40] proposed an adaptive fault-tolerant infrastructure, named Chameleon, which allows different levels of availability requirements in a networked environment, and enables multiple fault tolerance strategies including dual and TMR application execution modes.

The approach of design diversity, on the other hand, has mostly been applied in safety critical applications. The most famous applications of design diversity are the Boeing 777 airplane [35] and AIRBUS A320/A330/A340 aircraft [87]. The Boeing 777 primary flight control computer is a triple-triple configuration of three identical channels, each composed of three redundant computation lanes. Software diversity was achieved by using different programming languages targeting different lane processors. In the AIRBUS A320 series flight control computer [87], software systems are designed by independent design teams to reduce common design errors. Forced diversity rules are adopted in software development to ensure software reliability. In an experimental exploration of adopting design diversity in practical software systems, Popov and Strigini [72] implemented diverse off-the-shelf versions of relational database servers including Oracle, Microsoft SQL and Interbase databases in various ways.

The servers are distributed over multiple computers on a local network, on similar or diverse operating systems. The early results support the conjecture that reliability increases with the investment of design diversity.

2.5 Our Former Project Descriptions and Experimental Procedure

Motivated by the lack of empirical data, we conducted a real-world project for design diversity in the year 2002. The Redundant Strapped-Down Inertial Measurement Unit (RSDIMU) project involved more than one hundred students and 34 program versions were developed for a period of 12 weeks according to the same specification.

The specifications of a critical avionics instrument, Redundant Strapped-Down Inertial Measurement Unit (RS-DIMU), were used in our project investigation. RSDIMU was first engaged in [24] 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 2.10 show the system data flow diagram.

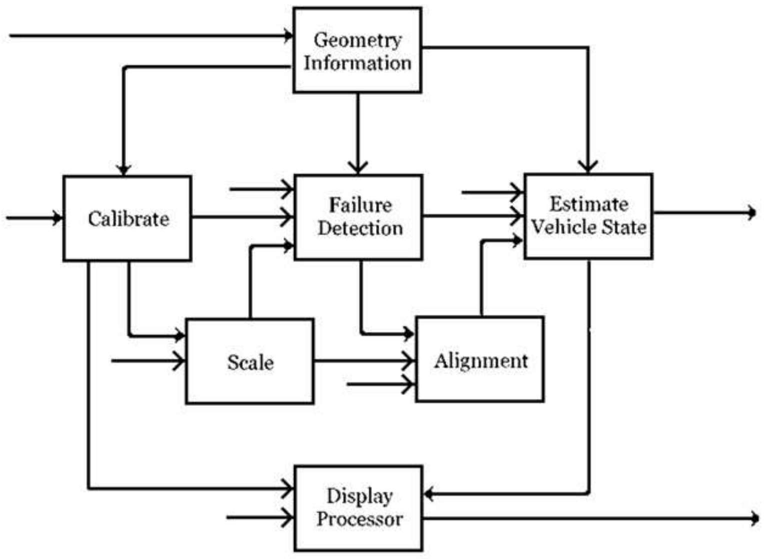

Figure 2.10. RSDIMU System Data Flow Diagram

In this project, eventually, 21 out of the 34 versions were selected to create mutants, each of which was injected with a single fault identified in the testing phase. Following a systematic rule for the mutant creation process, mutants were generated for testing and evaluation.

To investigate the nature and features of software failures, 1200 test cases were executed on these program *versions as well as the generated mutants for evaluation test. Based on these results, a number of analysis and evaluations were conducted, including fault classification and distribution, effectiveness of code coverage and mutant coverage, and the similarities between different mutants.*

The test cases conducted in the evaluation test is described in Table 2.2. Other details of the project and development procedures are discussed in out previous term paper and [64].

Table 2.2. Test case description

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.

Chapter 3

Fault Correlation Models

3.1 Evaluation on Popov, Strigini et al's Reliability Bounds Model

Popov, Strigini et al's model (PS model) [73] gave the upper and "likely" lower bounds for probability of failures on demand for a 1-out-of-2 diverse system. To get these bounds, complete knowledge on the whole demand space should be provided. As it is hard to obtain such knowledge, the demand space can be partitioned into some disjoint subsets, which are called subdomains. Given the knowledge on subdomains, failure probabilities of the whole system can be estimated as a function of the subdomain to which a demand belongs. The main idea is as follows.

For each subdomain S_i ($i = 1, \cdots, n$), we assume that the following probabilities are known: The probability $P(S_i)$ of a random demand during software operation being drawn from S_i and the probabilities of failure (pfds) of A and B ($P_{A,B|S_i}$) for demands from S_i, $P_{A|S_i}$ and $P_{B|S_i}$. Then

$$P_{A,B|S_i} = P_{A|S_i} P_{B|S_i} + cov_i(\Omega_A, \Omega_B). \tag{3.1}$$

The upper bound on the probability of system failure is determined as a weighted sum of upper bounds within subdomains:

$$P_{(A,B)} \leq \sum_i \min{(P_{A|S_i}, P_{B|S_i})} P(S_i). \tag{3.2}$$

The "likely" lower bound can be drawn from the assumption of conditional independence:

$$P_{A,B_{sub-ind}} = \sum_i P_{A|S_i} P_{B|S_i} P(S_i), \tag{3.3}$$

where $P_{A,B_{sub-ind}}$ is the actual probability of coincident failures in each subdomain if the versions fail independently.

Table 3.1. Alternative expressions for the pfd of a 1-out-of-2 system (from [73])

$\sum_{x \in D} \omega_A(x) \cdot \omega_B(x) \cdot P(x)$				
$P_A \cdot P_B$ (would be *pfd* in case of independence)	+	$cov(\Omega_A, \Omega_B)$ (accounts for variation of score between individual demands)		
$P_A \cdot P_B$	+	$cov(P_{A\|S_i}, P_{B\|S_i})$ (term for variation of *pfd* between subdomains)	+	$E(cov_i(\Omega_A, \Omega_B))$ (term for variation of score within each subdomain)
$\sum_i P_{A\|S_i} P_{B\|S_i} P(S_i)$ (*pfd* in case of independence in each subdomain)			+	$E(cov_i(\Omega_A, \Omega_B))$
$P_{A,B_{sub-ind}}$			+	$E(cov_i(\Omega_A, \Omega_B))$

Alternative expressions for $P_{A,B}$ as the pfd of a 1-out-of-2 version system are given in Figure 3.1.

This model can be applied to real-world data collected for diverse software. The upper bound and the lower bound can be estimated for applications using Point Estimate method or Confidence Bounds method. In our experiment, we adopt Point Estimate method to illustrate the modeling results.

3.1.1 Prediction Results Using Our Data Set

In our experiment, we created 426 mutants from 21 program versions, where each mutant was injected with one real fault into the final program versions passing the qualification test. Note the meaning of a mutant is different from that of a version, in the sense that a mutant is not a real final version but with faults injected manually. Here we treat each mutant, which contains only one real programming fault as a real version. From the analysis of severity of different faults, we notice that some faults can be more severe or even critical for the whole program, while others may have little influence on the program functionality. In this experiment, we only engage those mutants which passed the first 800 test cases [1] *(as a qualification test set) to study the failure correlation of the diverse versions.*

The RSDIMU application receives input values from redundant sensors and produces a consensus inertial measurement for avionic vehicles. The input domain for RSDIMU can be represented by various sensor failure conditions. In order to get the disjoint subdomains on the demand space, we follow the method described in [24] by dividing the 1200 test cases into 7 categories, i.e., $S_{0,0}$, $S_{0,1}$, $S_{1,0}$, $S_{1,1}$, $S_{2,0}$, $S_{2,1}$ and "others." These

[1] Out of the 1200 test cases conducted during qualification test, the first 800 test cases were designed to test various functionality of the application, while the last 400 test cases were randomly generated according to real operational scenarios.

categories (or so-called "states") denote different situations that the number of faulty sensors prior to or during the measurement operations. For example, $S_{1,0}$, indicates the "state" of the environment with a single faulty sensor prior to testing and no more sensor failures during the testing. We add the 7th state, i.e., "others" to denote the situations other than the above 6 operational states. It represents those test cases in which the whole RSDIMU system would fail under some extreme circumstances. Although it is indicated in [24] that such situation has little chance of occurring in mission-critical diverse systems, we still consider it as a subdomain of the total test cases due to the following reasons: 1) these seven disjoint subdomains compose the whole demand space which cannot be fully represented with only six states; 2) for reliable systems, the diverse versions need to react correctly to extreme situations.

As stated, we use the first 800 test cases as the qualification test. All the mutants which passed the qualification test are adopted in this experiment, and each mutant is treated as a single version. We apply the remaining 400 test cases on these selected mutants. The number of failures of these mutants (belonging to different versions) with respect to the states of test case are listed in Table 3.2. Note that the six mutants are from different initial versions with injection of different design and programming faults.

Table 3.2. Failure data of mutants passing qualification test

Mutant ID	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}
117	0	0	0	0	0	0	3
215	0	0	0	1	0	0	0
223	0	0	0	0	0	0	3
305	2	1	2	0	0	0	0
382	0	0	0	8	0	0	1
403	0	0	0	0	0	0	3

To apply PS model, we define the hypothetical demand profiles for calculation and illustrate the effect of the demand profile on the upper bounds and lower bounds. The adjusted demand profile is shown in Table 3.3. The former three in Table 3.3 are hypothetical demand files described in [73], while the last one (DP4) is the real probability distribution in our 400 test cases. Furthermore, in order to simulate the model more accurately and realistically, we select mutants belonging to different program versions, e.g., pair (117,305), (215,382) and (382,403). We adopt Demand Profile 4 in our analysis, which is the real probability distribution in our experiment.

In [73], Popov, Strigini et al discuss the use of both observed frequencies and of conservative confidence bounds as estimates of the conditional pfds, and favour the second alternative. Particularly in our case, according to Table 3.2, no failure was observed in some subdomains. Thus we adopt confidence bounds method to estimate the joint pfds in our experiment. Table 3.4 shows the 90% confidence upper bounds on pfds of mutants in subdomains, and

Table 3.3. Demand Profile

	DP1	DP2	DP3	DP4
$p(s_{0,0})$	0.99	0.4	0.15	0.4
$p(s_{0,1})$	0.005	0.2	0.15	0.1175
$p(s_{1,0})$	0.003	0.2	0.15	0.14
$p(s_{1,1})$	0.001	0.1	0.15	0.085
$p(s_{2,0})$	0.0005	0.05	0.15	0.0825
$p(s_{2,1})$	0.0003	0.03	0.15	0.0275
p_{others}	0.0002	0.02	0.10	0.1475

Table 3.6 displays the lower bounds. Our testing results for upper bounds and lower bounds on joint pfds under four demand profiles are listed in Table 3.5 and Table 3.7, respectively.

Table 3.4. 90 percent confidence upper bounds on mutants' pfds in subdomains

Mutant ID	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}
117	0.0142	0.0468	0.0396	0.0637	0.0655	0.1746	0.1080
215	0.0142	0.0468	0.0396	0.1066	0.0655	0.1746	0.0376
223	0.0142	0.0468	0.0396	0.0637	0.0655	0.1746	0.1080
305	0.0327	0.0786	0.0907	0.0637	0.0655	0.1746	0.0376
382	0.0142	0.0468	0.0396	0.3446	0.0655	0.1746	0.0633
403	0.0142	0.0468	0.0396	0.0637	0.0655	0.1746	0.1080

3.1.2 Comparison and Discussion

The target objects engaged in our experiment and NASA 4-university experiment studied in [73] are different. In the latter, diverse versions are employed to explore the granularity of failure correlations between different pairs of versions. But in our experiment, we treat mutants as the target diverse versions, and we know the exact fault each mutant contains. This is more helpful in finding realistic features of faults and their coincidence in diverse systems. Furthermore, to make the comparison more reasonable, we only test the mutants passing the qualification test and then capture their behavior in the subsequent operation testing. For better realism, i.e., similarity with real-world multiple-version systems, we select mutants derived from different program versions.

The behavior of three pairs of mutants show three different features of fault coincidence of design diversity. For pair (117, 305), the two mutants fail differently on the seven subdomains. In this case, $P_{117,305upper}$ is tighter

Table 3.5. Upper bounds on the joint pfds under Demand Profiles

Pair		$P_{117\_90\%}$	$P_{305\_90\%}$	$\min(P_{117\_90\%}, P_{305\_90\%})$	$P_{117,305_{upper90\%}}$
(117, 305)	DP1	0.0146	0.0332	0.0146	0.0146
	DP2	0.0400	0.0626	0.0400	0.0386
	DP3	0.0715	0.0796	0.0715	0.0644
	DP4	0.0483	0.0562	0.0483	0.0379
		$P_{215\_90\%}$	$P_{382\_90\%}$	$\min(P_{215\_90\%}, P_{382\_90\%})$	$P_{215,382_{upper90\%}}$
(215, 382)	DP1	0.0146	0.0149	0.0146	0.0146
	DP2	0.0429	0.0672	0.0429	0.0429
	DP3	0.0709	0.1091	0.0709	0.0709
	DP4	0.0415	0.0656	0.0415	0.0415
		$P_{382\_90\%}$	$P_{403\_90\%}$	$\min(P_{382\_90\%}, P_{403\_90\%})$	$P_{382,403_{upper90\%}}$
(382, 403)	DP1	0.0149	0.0146	0.0146	0.0146
	DP2	0.0672	0.0400	0.0400	0.0391
	DP3	0.1091	0.0715	0.0715	0.0670
	DP4	0.0656	0.0483	0.0483	0.0417

Table 3.6. 90 percent confidence lower bounds on mutants' pfds in subdomains

Mutant ID	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}
117	0.00065	0.00219	0.00185	0.00301	0.00309	0.00874	0.02939
215	0.00065	0.00219	0.00185	0.01529	0.00309	0.00874	0.00175
223	0.00065	0.00219	0.00185	0.00301	0.00309	0.00874	0.02939
305	0.00686	0.01113	0.01949	0.00301	0.00309	0.00874	0.00175
382	0.00065	0.00219	0.00185	0.16154	0.00309	0.00874	0.00890
403	0.00065	0.00219	0.00185	0.00301	0.00309	0.00874	0.02939

(smaller) than $min(P_{117}, P_{305})$ consistently for all demand profiles, although the difference between the two are insignificant under DP1. The reason behind is that the subdomains where mutant 117 performs better are those where mutant 305 performs worse, and vice versa, consistently. As the behavior of the two mutants are different in all subdomains, the covariance shown in Table 3.7 is a small positive number under DP1, while negative in the other three demand profiles. Thus the "likely" lower bound $P_{117,305_{sub\_ind10\%}}$ is greater than $P_{117} * P_{305}$ under DP1, but smaller under DP2, DP3 and DP4.

For the second pair of mutants (215,382), the covariance is positive under all demand profiles, indicating that the two mutants have related faults and may fail at the same subdomains. The upper bound $P_{215,382_{upper}}$ equals to $min(P_{215}, P_{382})$ under all demand profiles, since mutant 382 performs worse than mutant 215 in all subdomains. As the correlation between the two mutants, the lower bounds with 90% confidence are always tighter (greater)

Table 3.7. Lower bounds on the joint pfds under Demand Profiles

Pair		$P_{117\_10\%}$	$P_{305\_10\%}$	$cov(S_{117\_10\%}, S_{305\_10\%})$	$P_{117\_10\%}P_{305\_10\%}$	$P_{117,305_{sub\_ind10\%}}$
(117, 305)	DP1	$6.73 \cdot 10^{-4}$	$6.91 \cdot 10^{-3}$	$3.86 \cdot 10^{-8}$	$4.65 \cdot 10^{-6}$	$4.69 \cdot 10^{-6}$
	DP2	$2.37 \cdot 10^{-3}$	$9.62 \cdot 10^{-3}$	$-4.26 \cdot 10^{-6}$	$2.28 \cdot 10^{-5}$	$1.86 \cdot 10^{-5}$
	DP3	$5.87 \cdot 10^{-3}$	$8.02 \cdot 10^{-3}$	$-1.80 \cdot 10^{-5}$	$4.71 \cdot 10^{-5}$	$2.91 \cdot 10^{-5}$
	DP4	$5.87 \cdot 10^{-3}$	$7.79 \cdot 10^{-3}$	$-2.47 \cdot 10^{-5}$	$4.57 \cdot 10^{-5}$	$2.09 \cdot 10^{-5}$
		$P_{215\_10\%}$	$P_{382\_10\%}$	$cov(S_{215\_10\%}, S_{382\_10\%})$	$P_{215\_10\%}P_{382\_10\%}$	$P_{215,382_{sub\_ind10\%}}$
(215, 382)	DP1	$6.80 \cdot 10^{-4}$	$8.27 \cdot 10^{-4}$	$2.39 \cdot 10^{-6}$	$5.26 \cdot 10^{-7}$	$2.95 \cdot 10^{-6}$
	DP2	$3.05 \cdot 10^{-3}$	$1.78 \cdot 10^{-2}$	$1.98 \cdot 10^{-4}$	$5.43 \cdot 10^{-5}$	$2.52 \cdot 10^{-4}$
	DP3	$4.95 \cdot 10^{-3}$	$2.76 \cdot 10^{-2}$	$2.50 \cdot 10^{-4}$	$1.37 \cdot 10^{-4}$	$3.86 \cdot 10^{-4}$
	DP4	$2.83 \cdot 10^{-3}$	$1.63 \cdot 10^{-2}$	$1.70 \cdot 10^{-4}$	$4.62 \cdot 10^{-5}$	$2.16 \cdot 10^{-4}$
		$P_{382\_10\%}$	$P_{403\_10\%}$	$cov(S_{382\_10\%}, S_{403\_10\%})$	$P_{215\_10\%}P_{382\_10\%}$	$P_{382,403_{sub\_ind10\%}}$
(382, 403)	DP1	$8.27 \cdot 10^{-4}$	$6.73 \cdot 10^{-4}$	$4.62 \cdot 10^{-7}$	$5.57 \cdot 10^{-7}$	$1.02 \cdot 10^{-6}$
	DP2	$1.78 \cdot 10^{-2}$	$2.37 \cdot 10^{-3}$	$1.61 \cdot 10^{-5}$	$4.23 \cdot 10^{-5}$	$5.84 \cdot 10^{-5}$
	DP3	$2.76 \cdot 10^{-2}$	$5.87 \cdot 10^{-3}$	$-4.86 \cdot 10^{-5}$	$1.62 \cdot 10^{-4}$	$1.13 \cdot 10^{-4}$
	DP4	$1.63 \cdot 10^{-2}$	$5.86 \cdot 10^{-3}$	$-1.16 \cdot 10^{-5}$	$9.56 \cdot 10^{-5}$	$8.40 \cdot 10^{-5}$

than $P_{215} * P_{382}$ under all subdomains. This positive covariance case supports the concept of "variation of difficulty" between and within different demand subdomains.

The third pair (382,403) shows the possibility of negative covariance on DP3 and DP4. The covariance is a small negative number, and thus the lower bound is smaller than the probability under independence scenario. It indicates that with design diversity, the covariance of different versions may become a benefit instead of a disadvantage. Nevertheless, as in [73], our data also show that this situation is less likely to happen under DP1. The reason behind may be that the two mutants have correlations on some subdomains and no correlation on other subdomains, i.e., they have coincident failures on S_{others}, but no coincident failures on $S_{1,1}$. In DP1, the probability of the "independence" subdomain $S_{1,1}$ is a small number; while in other three demand profiles, the probability of $S_{1,1}$ is large enough to affect the overall correlation and make the reliability even higher than that of assuming "independence".

In order to assess whether the approach proposed in [73] is useful in practice, we need to answer the following questions:

1. "Does this method always produce tighter bounds than $P_A * P_B$ and $min(P_A, P_B)$?" From the analysis and discussion above, we can see that the confidence bounds are tighter under most circumstances except two situations: 1) one mutant performs worse than the other in all subdomains; and 2) with negative covariance, the lower bound is smaller than the probability under independent scenario.

2. "Does this method give tight enough predictions when used in practice?" To this question, we cannot give answers on the basis of our data, since in our experiment probabilities of common failure are measured directly from the number of common failures observed. The original method in [73] is meant for cases in which one can obtain estimates of failure probabilities (per subdomain) for the two versions separately, but does not have a chance of observing the two versions on the same test cases before making a prediction. Further experimental data are needed to be explored to answer this question.

Overall, the approach proposed in [73] of analyzing the behaviors of the versions by subdomains appears to help, with our project data, in revealing the features of failure correlation among diverse programs.

3.2 Evaluation on Dugan and Lyu's System Reliability Model

Dugan and Lyu (DL model) proposed a dependability modeling methodology for fault-tolerant software and systems [23]. The DL reliability model is constructed by three parts: a Markov model details the system structure, and two fault trees represent the causes of unacceptable results in the initial configuration and in the reconfigured degraded state. Based on this 3-level reliability model, three parameters can be estimated according to the experimental data: P_V*, the probability of an unrelated fault in a version;* P_{RV}*, the probability of a related fault between two versions; and* P_{RALL}*, the probability of a related fault in all versions. The fault tree models for 2, 3 and 4 version systems are shown in Figure 3.1. The three parameters are calculated by the following equations for 3-version systems:*

$$P_V = \frac{F_1}{NF_0 + F_1}, \tag{3.4}$$

$$P_{RV} = \frac{2F_2 P_V(1-P_V) - (N-1)F_1 P_V^2}{2F_2 P_V(1-P_V) + (N-1)F_1(1-P_V^2)}, \tag{3.5}$$

$$P_{RALL} = \frac{F_3 - {P_V}^3}{1 - {P_V}^3}, \tag{3.6}$$

where F_1,F_2 *and* F_3 *represent the observed frequency of a single failure, two and three coincident failures, respectively, in a 3-version configuration.*

In order to verify the effectiveness and consistency of DL model, we apply new data to this model and compare our results with original results in [23]. In this experiment, we employ the same six mutants passing the qualification test as the target versions in this fault tree model and their failure characteristics are investigated. The 400 operational test cases were executed on these mutants and the failures encounter in each mutant are shown

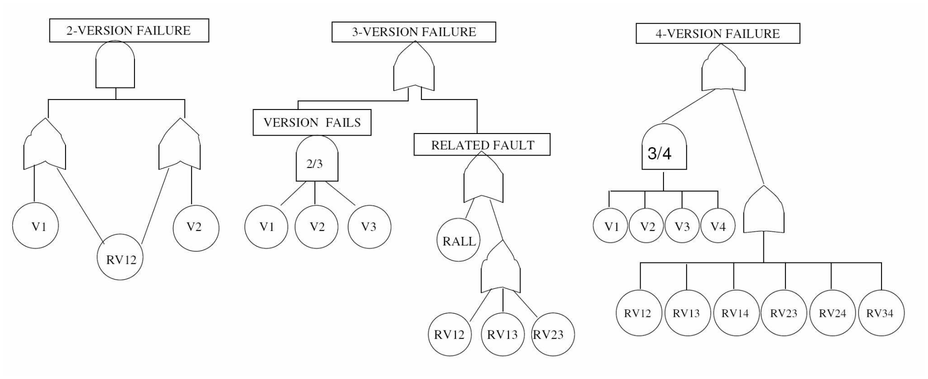

Figure 3.1. Fault tree models for 2, 3 and 4 version systems (from [23])

in Table 3.8. We can see from Table 3.8 that the average failure probability for single version is 0.01, which is much smaller compared with the original experimental data in [23]. It indicates that the versions we used in this experiment is more reliable. Moreover, the small failure frequency does not affect the prediction accuracy in terms of magnitude.

Table 3.8. failure Characteristics for individual mutants

Mutant ID	Number of failures	Prob. By-case
117	3	0.0075
215	1	0.0025
223	3	0.0075
305	5	0.0125
382	9	0.0225
403	3	0.0075
Average	4	0.01

We configure the six mutants in pairs, and compare their outputs for each test case. Table 3.9 yields an estimate of $P_V = 0.0084$ *for the probability of raising an unrelated failure in a 2-version configuration, and an estimate* $P_{RV} = 0.0016$ *for the probability of a related failure.*

Table 3.9. failure Characteristics for 2-version configurations

Category	Number of cases	Frequency
F0 - no failure	5890	0.9817
F1 - single failure	100	0.0167
F2 - two coincident	10	0.0017
Total	6000	1.0000

Next, the six mutants are configured in sets of three. Table 3.10 shows the number of times that 0, 1, 2 and 3 failures occurred in the 3-version configuration. The data yields an estimate of $P_V = 0.0071$ *for the probability of an independent failure. The comparison between the predicted probability of 0, 1, 2 and 3 failures using independence model and observed frequency are shown Table 3.11. Unlike the previous experiment reported in [22], our data shows that the observed frequency for two and three simultaneous failures is higher than that of the independence model. The data also yields the estimation of* $P_{RV} = 0.0013$ *for the probability of two related failures, and* $P_{RALL} = 0.0004$ *for the probability of failures involving all three versions.*

Table 3.10. failure Characteristics for 3-version configurations

Category	Number of cases	Frequency
F0 - no failure	7797	0.9746
F1 - single failure	169	0.0211
F2 - two coincident	31	0.0039
F3 - three coincident	3	0.0004
Total	8000	1.0000

Table 3.11. Comparison of independent model with observed data for 3 versions

No. of failures	Independent model	Observed frequency
0	0.9786	0.9746
1	0.0213	0.0211
2	0.0002	0.0039
3	0	0.0004

The mutants are then analyzed in combinations of four programs. Table 3.12 shows the number of times that 0, 1, 2, 3 and 4 failures occurring in the 4-version configuration. The data yields an estimate of $P_V = 0.0063$ *for the probability of an independent failure. The comparison between the predicted probability of 0, 1, 2, 3 and 4 failures using independence model and observed frequency are shown in Table 3.13. Just like 3-version configuration, our data shows that the observed frequency for three and four coincident failures is higher than that of the independence model. The data also yields the estimation of* $P_{RV} = 0.0028$ *for the probability of two related faults, and* $P_{RALL} = 0$ *for the probability of coincident failures in all four versions.*

Table 3.14 summarizes the parameters estimated from our data. The parameters are applied to the fault tree model shown in Figure 3.1. The predicted system failure probability using derived parameters in the fault tree models agrees quite well with the observed data, especially with the 2- and 3-version configurations. For the 4-version configuration, the predicted probability is close to zero but the observed frequency is 0.0015. Our

Table 3.12. failure Characteristics for 4-version configurations

Category	Number of cases	Frequency
F0 - no failure	5811	0.9685
F1 - single failure	147	0.0245
F2 - two coincident	33	0.0055
F3 - three coincident	9	0.0015
F4 - four coincident	0	0.0000
Total	6000	1.0000

Table 3.13. Comparison of independent model with observed data for 4 versions

No. of failures	Independent model	Observed frequency
0	0.9750	0.9685
1	0.0247	0.0245
2	0.0002	0.0055
3	0	0.0015
4	0	0

experiment shows that the predicted system failure probability from fault tree model is very close to the observed values in most situations, except that there is a gap between the two in 4-version model. This should be further investigated to validate the effectiveness and accuracy of the fault tree model.

Table 3.14. Summary of parameter values derived from our data

2-version model	3-version model	4-version model
P_V = 0.0084	P_V=0.0072	P_V=0.0063
P_{RV} = 0.0016	P_{RV} = 0.0013	P_{RV}=0.0028
	P_{RALL}= 0.0004	P_{RALL}= 0
Predicted system failure probability (from the model)		
0.0017	0.0045	0.000048
Predicted system failure probability (from the data)		
0.0017	0.0043	0.0015

Figure 3.2 compares the predicted reliability of three different configurations, including 2-version configuration for Distributed Recovery Block (DRB) [76], 3-version configuration for N-Version Programming (NVP) [3, 61], and 4-version configuration for N-Self Checking Programming (NSCP) [48]. We can see from our experiment that DRB is the most reliable of the three to produce a correct result, while NSCP is the least reliable. Compared with the original experimental data in [22], the prediction performance of the three configurations in our experiment

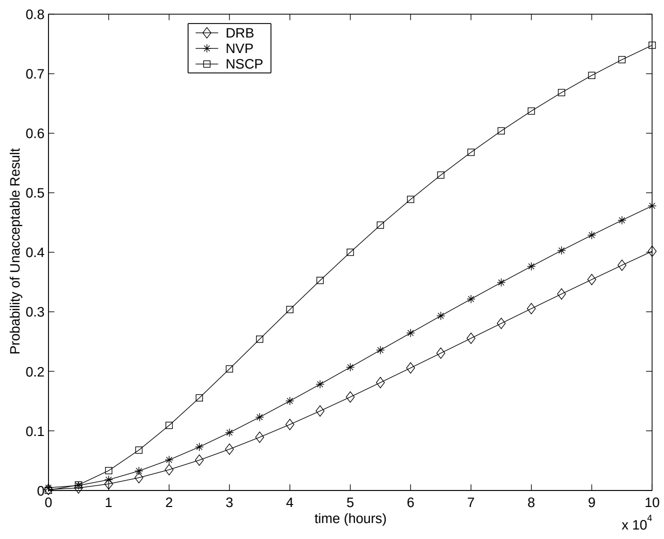

Figure 3.2. Predicted reliability by different configurations

are consistent with those in [22]. However, if we look into the first hundreds of hours, the three configurations performs differently, as shown Figure 3.3. Here NSCP depicts higher reliability than DRB and NVP, although it gives the least reliability in the long run.

Figure 3.4 compares the predicted safety of the three systems. Here we assume that the decider used in the NVP and NSCP has a failure probability of only 0.0001 and that for DRB has a failure rate of 0.001 [23]. According to Figure 3.4, NSCP is the most likely to produce a safe result, while DRB are an order of magnitude less safe than NSCP. This is also consistent with the original experimental results in [23].

Overall, compared our project with former project in [23], the reliability and safety performance of DRB, NVP, NSCP shows consistency of DL model with respect to our experimental data. The discrepancy in the first hundreds of hours may indicate dependence on operational domains and needs further investigations. Furthermore, the above predictions are on the basis of our primary data, some assumptions in [23] and the fault tree modeling. To achieve more accurate results, the information about the correlation between successive executions should be included [80].

3.3 Summary

In this chapter, we perform analysis and investigation on reliability and fault correlation issues for diverse software systems. We apply our RSDIMU project data to evaluate the effectiveness and prediction accuracy of existing reliability models for fault tolerant software. In our experiment, mutants with real faults are engaged. 400 operational test cases were executed on six mutants which passed a qualification test to investigate the fault

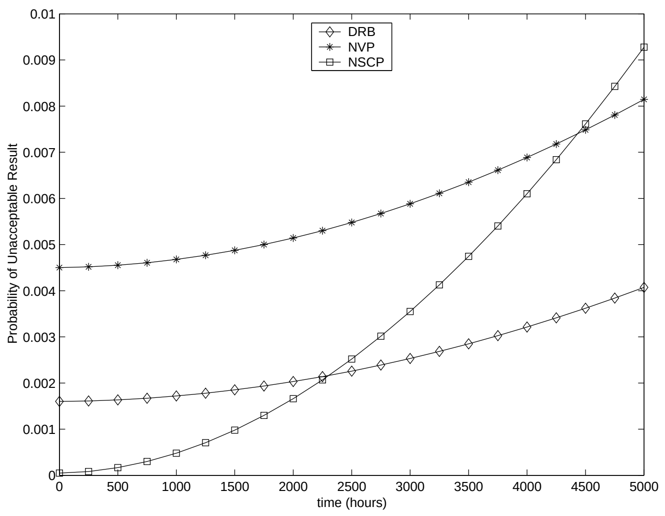

Figure 3.3. Predicted reliability by different configurations

correlation features between any pairs of mutants.

We first apply Popov, Strigini et al's reliability bounds model to locate the upper and lower bounds for reliability of diverse programs. The results reveal that the confidence bounds are tighter with our data in most situations. It verifies the hypothesis of "variety of difficulties" on different demand subdomains, and supports the effectiveness of design diversity with small fraction of positive fault correlations and existence of negative correlations. Furthermore, we adopt Dugan and Lyu's dependability model to parameterize the reliability of different configurations. The analysis shows that NSCP is the least reliable but most safe approach among the three, while DRB inherits the highest reliabililty but the lowest safety according to our experimental data in the long run. The discrepancies in the first hundreds of hours may relate to the operational domain and needs further investigation.

As our future work, we will further analyze the prediction accuracy of these reliability and fault correlation models on design diversity. Other comprehensive models such as Tomek and Trivedi's model using stochastic reward nets will be parameterized and analyzed by our data set. Further testing and verification will be explored on our data set to collect more experimental results.

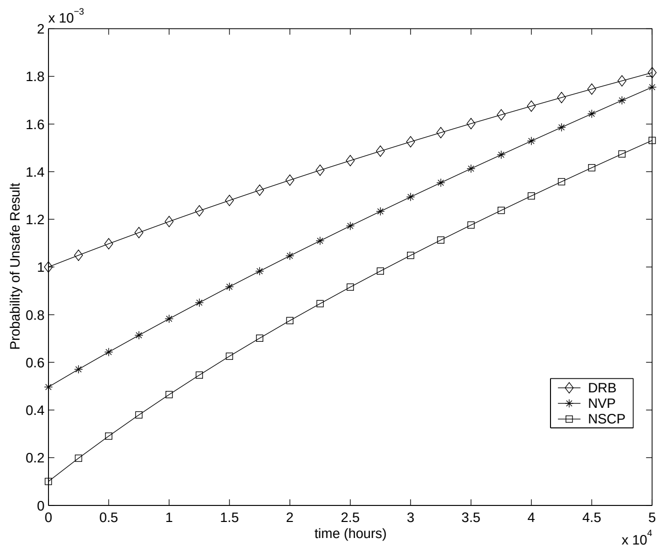

Figure 3.4. Predicted safety by different configurations

Chapter 4

The Effectiveness of Code Coverage

4.1 Effectiveness of Code Coverage in Different Testing Profiles

As we have mentioned above, the relationship between code coverage and fault coverage is very complicated according to former empirical observations. The correlation between the two measures varies in different experiments, thus causing the question on whether a causal-effect dependency exists in code coverage and reliability. Both theoretical insight and empirical data are needed to clarity this question.

As code coverage is measured as the portion of program code, which is defined by different coverage criterion. The four popularly used code coverage criteria are: block coverage, decision coverage, C-use and P-use. The definitions for different coverage are given in [36]. We give brief descriptions of each as follows:

Block coverage is measured as the portion of basic blocks executed. Basic blocks are maximal code fragment without branching, containing no internal flow of control change;

Decision coverage is measured as the portion of decisions executed. A decision is a code fragment associated with a branch predicate.

C-use coverage is measured by computational uses covered. It refers to a pair of definition and computational use of a variable.

P-use coverage is measured by predicate uses covered. It refers to a pair of definition and predicate use of a variable.

From the definitions of these four coverage metrics, block coverage and C-use contain no control flow change while decision coverage and P-use are related to branch predicates.

According to previous work from others and ourselves, we notice that the effect of code coverage on fault coverage is positive in general. However, this correlation varies a lot in different reports. The intuitive reason of *using code coverage as an indicator for software reliability is that code constructs not exercised during test may contain faults. But considering the requirements in specification, on the one hand, test set with additional code coverage is more effective in detecting faults; on the other hand, some test cases with less code coverage can still detect more program faults, when new code fragments are exercised which are not covered by other test cases.*

Based on these considerations, we hypothesize that: 1) the effect of code coverage on fault detection varies if different testing profiles are examined; 2) different code coverage metrics may have influence on such correlation.

To investigate the above effect of different code coverage metrics under different testing profiles, empirical data are seriously needed. It requires a software project with bug history recorded, so that real faults can be studied, code coverage can be measured, test effectiveness can be quantified and test cases can be analyzed. Moreover, in such experiment, the development process should be controlled, the population of program versions should be large enough, and the application should be complicated as real-world projects in practice to ensure the software complexity.

Motivated by the lack of experimental data satisfying the requirement above, we conducted a experiment adopting the RSDIMU avionics application [64]. The application was part of the navigation system in an aircraft or spacecraft, and was first engaged in [24] for NASA-sponsored 4-university multi-version software project.

We employ mutation testing in our investigation. Mutation testing is one of the main schemes for test case selection and evaluation [71]. It starts with creating many versions of a program. Each version is "mutated" to introduce a single fault. These "mutant" programs are run against test cases with the purpose of causing each faulty version to fail. Each time when a test case causes a faulty version to fail, the mutant is considered "killed". An effective test case always kills more mutants than a less effective test case does.

Based on the detected software faults, we selected 21 program versions and created 426 software mutants, and conducted coverage testing [63] and mutation testing [71]. 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 4.1. The decision, C-use and P-use coverage measures expose almost exactly the same pattern except for their absolute values, and thus omitted here.

The contribution of each test case in covering (killing) the mutant population is shown in Figure 4.2. Figure 4.1 and Figure 4.2 clearly portray certain patterns between block coverage and fault detection under six different test profiles, as delimited by A-E in the figures. On the one hand, test coverage and mutant coverage show similar capability in revealing patterns in the test cases. On the other hand, higher and more stable code coverage, e.g., that achieved by test cases 1001-1200, may result in lower and unstable fault coverage.

For the overall test set, the former 800 test cases are designed according to the specification, which are named *as functional testing. To latter randomly generated 400 test cases are so-called random testing.*

Other detailed descriptions of the test set as well as the experiment can be found in [64].

4.2 Experimental Evaluation and Testing Results

Based on our former experimental data, we further explore the relationship between code coverage and fault detection capability for the current 1200 test cases which fall into six regions according to the various patterns revealed in Figure 4.1 and Figure 4.2. As described above, these test cases can be classified as functional testing (1-800) and random testing (801-1200). They can also be categorized by the system status: normal operation testing and exceptional operation testing. In this study, we examine the relationship in all these classifications and survey their similarities and differences.

To answer the question: Is code coverage a good indicator of fault detection capability? We investigate the statistical relationship between code coverage and fault coverage using linear regression model. In our experiment, each mutant stands for one real fault in the software development process. Thus the terms "fault" and "mutant" are used interchangeably in this paper.

In the following, we will examine the different relationship on three aspects: 1) the situations in overall test set and different regions; 2) the situations in functional testing versus random testing; and 3) the situations in normal operational testing versus exceptional operational testing.

4.2.1 The relationship revealed in different test case regions

As mentioned before, the former 800 test cases were designed to target different functions of the system, and the latter 400 test cases were randomly generated to simulate the operational environment. Moreover, as shown in Figure 4.1 and Figure 4.2, block coverage and fault coverage show different patterns in different parts on the whole test set. We divide the whole test set into six regions according to their patterns (see Table 4.1). These six clusters also reflect the underlying design principles of different test cases. After applying linear regression model on current data, we get the parameters and the quality of fit of linear models in various regions as well as in the whole test case space, as illustrated in Table 4.1. The results show that the relationship between block coverage and mutant coverage can be predicted by linear model at the whole test case space with the value of R-square of 0.781 (see Figure 4.3). But as a measure of the quality of fit, R^2 ranges dramatically from 0.189 (in Region IV) to 0.98 (in Region VI) in different test case regions, as shown in Figure 4.4 and Figure 4.5.

Figure 4.3 indicates that code coverage is a moderate indicator for fault detection capability of given test cases. Generally, the larger of the code coverage that a test case executes, the more mutants it kills in program versions.

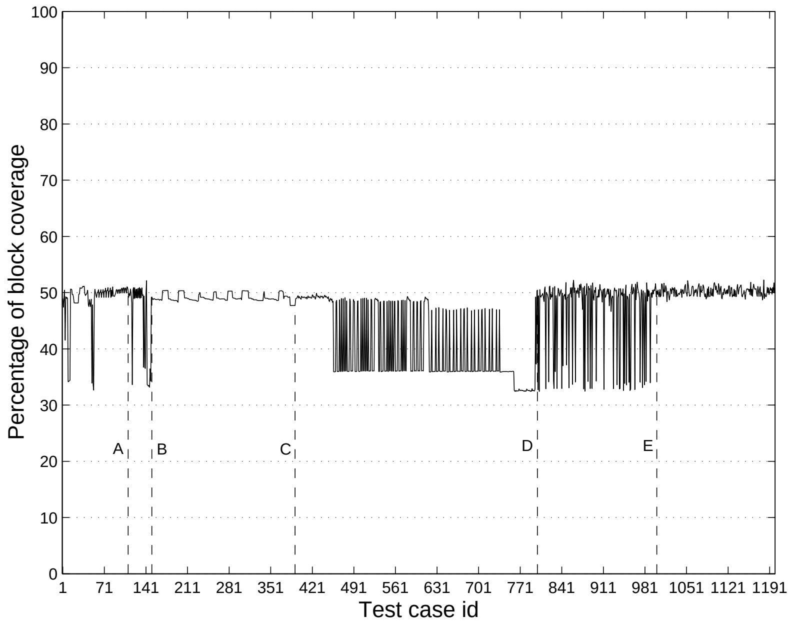

Figure 4.1. Test Case Contribution on Program Coverage

But different phenomenon can be observed if we view the whole figure as a combination of two clusters: one with block coverage at about 35% and mutant coverage at 90-150, and the other with block coverage at about 50% and mutant coverage at 150-270. In each cluster, the relationship between block coverage and mutant coverage is not always a positive correlation. Test cases with larger block coverage may kill less mutants, while test cases with smaller block coverage may cause more mutants to fail.

However, if we look into the linear regression relations between block coverage and mutant coverage in each of the six regions, we can find the most fit in Region IV and the least fit in Region VI. Note that test cases in Region IV

Table 4.1. Parameter and fitness of linear models in different test case regions

Test case region	R-square
Overall (1-1200)	0.781
Region I (1-111)	0.634
Region II (112-151)	0.724
Region III (152-392)	0.672
Region IV (393-800)	0.981
Region V (801-1000)	0.778
Region VI (1001-1200)	0.189

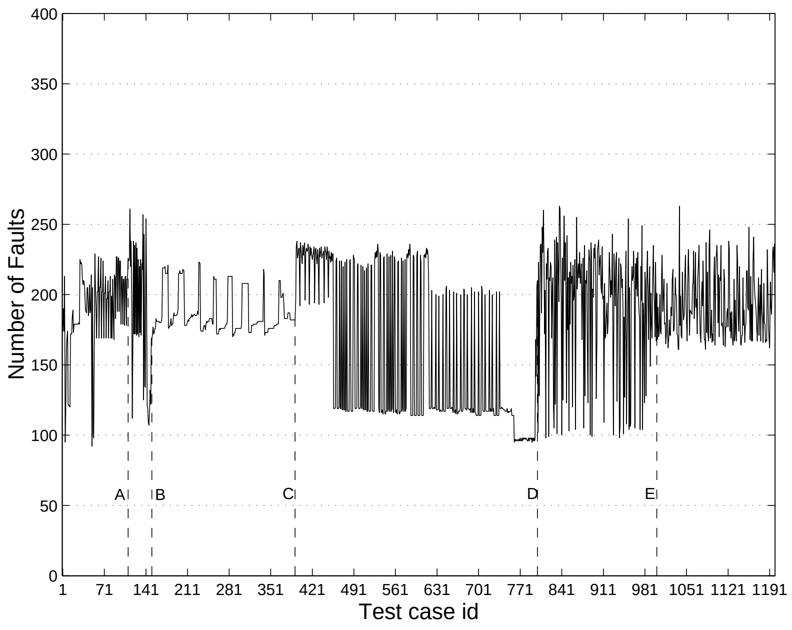

Figure 4.2. Test Case Contribution on Mutant Coverage

are designed with various combinations of the system status, while test cases in Region VI are randomly generated with a single initial random seed. One of the reasons behind this phenomenon may be because of the design principle of test cases in Region IV, which targets at the main control flow of the program. The more program code portion they execute, the more likely that program versions fail. This agrees with the traditional assumption and observation that more code coverage brings more fault coverage. The other reason lies that for Region VI, all the 200 test cases have very close block coverage (from 48% to 52%). It agrees with our earlier observation in two clusters: If the code coverage is in a small range, the linear correlation between code coverage and fault coverage may be insignificant. Furthermore, as shown in the latter analysis, we believe the strong correlation in Region IV lies in the fact that large number (277/373) of exceptional test cases contained in this region.

4.2.2 The relationship revealed in functional testing versus random testing

Functional testing and random testing are two basic methods employed in test case generation. In our test set, 800 test cases are functional test cases based on the basic operational requirements in the specification. The other 400 test cases are randomly generated with different seeds to simulate the large data set in real operations. The linear correlation in functional testing and random testing can be seen in Table 4.2. The correlation in functional testing is larger than that in random testing, but the difference is not significant. In general, functional test cases are designed to increase their code coverage (i.e., to cover more code fragments), while random test cases are

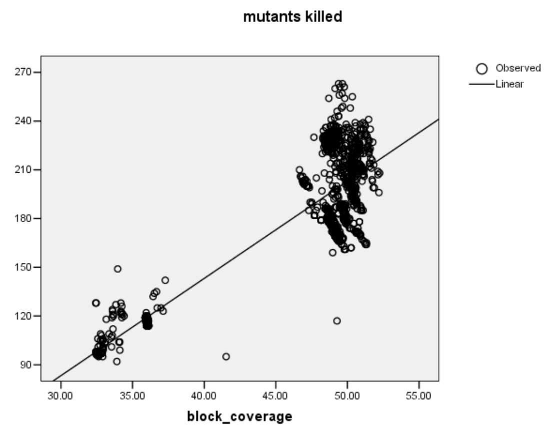

Figure 4.3. Linear Regression Relations between Block Coverage and Defect Coverage

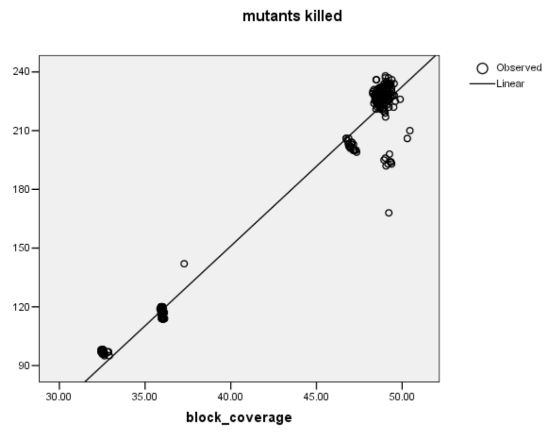

Figure 4.4. Linear Regression Relations between Block Coverage and Defect Coverage in Region IV

generated to simulate real operational environment and not likely to improve code coverage. From our results, some functional test cases inherit the strong linear correlation between code coverage and fault coverage (e.g., in Region IV), while some random test cases show little correlation between the two measures (e.g., in Region VI). The underlying reason may be that there is no exceptional test cases in Region VI, while a large number of exceptional test cases (277 in Region IV while 373 in total test set) in Region IV. For another random test region, i.e., Region V, positive correlation is also observed with R^2 0.778 *as there are 56 exceptional test cases in this region.*

However, in average, the correlations between code coverage and fault coverage vary from 0.837 in functional testing to 0.558 in random testing. In both situations, code coverage is a moderate indicator for fault detection capability.

The effectiveness of random testing has been a controversial [92]. For the question whether random testing is

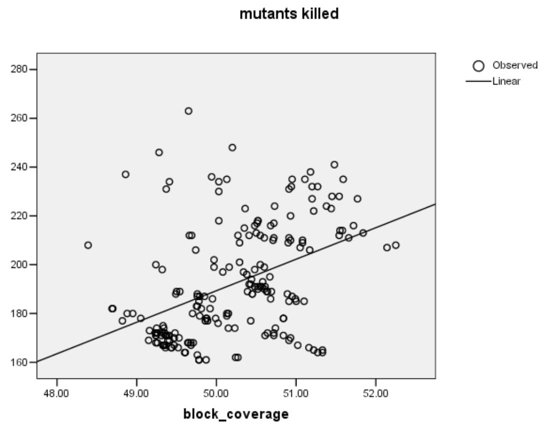

Figure 4.5. Linear Regression Relations between Block Coverage and Defect Coverage in Region VI

Table 4.2. R-square value in testing profiles

Testing profile(size)	R-square
Whole test set(1200)	0.781
Functional test(800)	0.837
Random test(400)	0.558
Normal test(827)	0.045
Exceptional test(373)	0.944

an effective testing approach, we can see some positive signs from our statistical data. First, although random test cases are not designed to improve code coverage, they can still achieve similar code coverage as those functional test cases, e.g., the similar code coverage (around 50%) obtained in Region VI as that in Region IV. Secondly, random test can kill mutants whose faults are hard to detect, i.e., with small number of failure occurrence. If we examine the failure details of mutants that failed at less than 20 test cases (which means these mutants inherit low failure occurrence), we find that there are 94 random test cases and 169 functional test cases that can detect these faults. The percentage 35.7% ($\frac{94}{94+169}$) shows that random test cases are effective to detect hard-to-kill mutants as well as functional test cases. The numbers and failure occurrence of mutants that failed in only functional testing as well as in random testing are listed in Table 4.3. The figures indicate that there are 382 mutants killed in functional testing and 371 mutants killed in random testing. Among all these mutants, 362 mutants failed at both testing, 20 mutants (with mean failure number of 4.5) killed by functional testing only and nine mutants (with 3.67 failures in average) failed at random test cases only. This means that random testing may miss 5.2% (20/382) faults compared with functional testing, but it kills 2.4% (9/371) additional faults which are not detected *by functional testing. These nine newly-killed mutants inherit pretty low failure occurrence.*

Table 4.3. The failure number of mutants that failed in different testing

Test case type	Mutants killed	Mean failure number	Std. deviation
Functional testing	20/382	4.50	3.606
Random testing	9/371	3.67	2.236
Normal testing	36/371	120.00	221.309
Exceptional testing	20/355	55.05	99.518

Overall, random testing is a necessary complement to functional testing. Code coverage is still a good indicator of fault detection capability for functional as well as random test cases.

4.2.3 The relationship revealed in normal operational testing versus exceptional testing

Test cases are designed to detect and remove residual faults in program versions which are developed to satisfy the requirements in the software specification. There are two major system status according to the specification: normal operation and exception handling. A test set should contain test cases designed according to these two system operation scenarios to hit all kinds of faults. The classification of normal and exceptional status is application-dependent and defined by the specification. Particularly in RSDIMU application, normal operations refer to those situations where at most two sensors fail as the input and at most one sensor fails during the test. All the other cases, which cause the difficult conditions where acceleration of the instrument unable to be estimated, are viewed as exceptional operations.

As shown in Table 4.2, the linear correlation of code coverage and fault coverage changes dramatically from normal testing (0.045) to exceptional testing (0.944). It clearly indicates the strong correlation of the two measures in exceptional testing, but no correlation in normal testing, as seen in Figure 4.6 and Figure 4.7, respectively.

In normal testing, the code coverage range is relatively small (see Figure 4.6), between 48% and 52%. This agrees with the design principle of normal test cases. The normal operations should execute the major part of the program versions. In such a situation, although high code coverage may be obtained, it cannot be employed to predict the fault detection capability of a normal operational test case. On the contrary, in the case of exceptional testing, the value of R-square of 0.944 indicates an obvious positive correlation between code coverage and fault coverage.

Figure 4.7 contains two main clusters. We examine the exceptional test cases and find that these two clusters

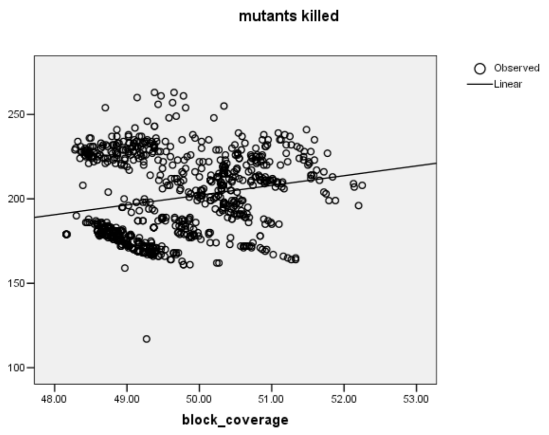

Figure 4.6. Linear Regression Relations between Block Coverage and Defect Coverage in normal testing

are caused by the specific application. Because of the complexity of the RSDIMU application, some functions such as acceleration estimation, contain large-scale computational code. In some exceptional cases, part of these functions can be executed but others be skipped (e.g., When four sensors on exactly two faces have failed before the test, and no additional sensor fails during the test); while in other cases, all these computational codes are skipped according to the system status. This explains why the code coverage shows two different ranges and a big gap exists between the two clusters. Although this phenomenon is application specific, the strong correlation pattern provides a positive support for the code coverage. We postulate that even in other applications, since different exceptional test cases simulate different exceptional situations, a variation of code coverage are achieved although the ranges of code coverage may be larger or smaller compared with our application. Test cases with higher code coverage are likely to detect more faults, i.e., the correlation between code coverage and fault coverage may still hold. Of course, this needs further empirical investigation.

According to Table 4.3, the mutants killed by exceptional testing only fail less frequently (with 55 failures in average) than those failed at normal testing only (with 120 failures in average). Considering the total numbers of test cases in normal testing and exceptional testing are 827 and 373, the normalized failure occurrences of these two classes of mutants are similar (120/827 vs. 55/373). Normal testing can detect more faults than exceptional testing (371 vs. 355), yet it contains larger test set than exceptional testing.

Table 4.3 also reveals that mean failure numbers under functional testing and random testing are significantly different from those under normal testing and exceptional testing. It may imply the different features and relationship among the four testing profiles. Functional testing (which designed according to the specification) and

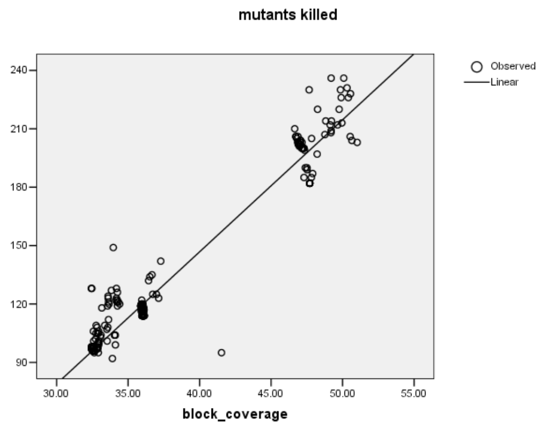

Figure 4.7. Linear Regression Relations between Block Coverage and Defect Coverage in exceptional testing

random testing (which designed according to operations) have a big overlap. Most cases under the two testing profiles can detect similar faults. Only a small amount of function-specific faults or faults under some extreme situations can be detected by functional testing or random testing only. But for normal testing and exceptional testing, the two testing profiles are parallel, i.e., they contain no overlap. A fault only occurring under normal operations may fail at many normal test cases, but it cannot be detected by exceptional testing, and vice versa. The different features and relationship among testing profiles can also explain the various patterns they inherit in terms of the correlation between code coverage and fault coverage: there is a similarity between functional testing and random testing, but a major difference between normal testing and exceptional testing.

In summary, both normal operational testing and exceptional testing are important for software testing. But code coverage is clearly a good indicator of fault detection capability of exceptional test cases, rather than normal test cases. This can also give some hints on designing the exceptional test cases: increasing the code coverage of such test cases will gain benefits on fault detection capability.

4.2.4 The relationship revealed in different combinations for various coverage metrics

From the data shown above, we observe that the effect of code coverage on fault coverage is significant in exceptional testing, while weak in normal testing. The difference between functional testing and random testing is not obvious, but still code coverage is a moderate indicator for test effectiveness. To further illustrate such effect, we examine the correlation pattern in different testing profile combinations. The linear regression fit in the four combinations are listed in Table 4.4. It is shown that the combinations containing exceptional testing *(random/exceptional and functional/ exceptional) indicate strong correlation, while the combinations containing normal testing (random/normal and functional/normal) inherit weak correlation.*

Table 4.4. Linear Regression Fitness for combinations

Testing Combination	R-Square
random & normal	0.045
random & exceptional	0.949
functional & normal	0.076
functional & exceptional	0.950

Table 4.5. R-square value in different code coverage and testing profiles

Testing profile(size)	block coverage	decision coverage	C-use	P-use
Whole test set(1200)	0.781	0.832	0.774	0.834
Functional test(800)	0.837	0.880	0.830	0.881
Random test(400)	0.558	0.646	0.547	0.648
Normal test(827)	0.045	0.368	0.019	0.398
Exceptional test(373)	0.944	0.952	0.954	0.954

To investigate the correlation pattern between different code coverage metrics and test effectiveness under various testing profiles, the R-square values of linear regression for decision coverage, C-use and P-use are listed in Table 4.5, compared with that of block coverage. The other three coverage metrics show similar patterns as block coverage. There is an insignificant difference between block coverage/C-use and decision coverage/P-use under normal testing. One possible reason may be that the variation of decision coverage and P-use coverage are larger under normal operations, as they are related to the control flow change in the program code. According to our previous observation, larger variation in code coverage implies more correlation in terms of the relationship among different clusters.

4.3 Discussions

Based on our project data, we investigate the effect of different code coverage metrics under different testing profiles. we focus on the following two questions: 1) Does the effect of code coverage on fault detection vary under different testing profiles? 2) Do different code coverage metrics have various effects on such relationship?

For the first question, based on above experimental data, our answer is supportive. The correlation varies under different testing profiles. In particular, there is a significant correlation between code coverage and fault detection capability for exceptional test cases. Positive linear correlation holds with an overall R-square of 0.944. The relationship shows no correlation for normal operational test cases. The phenomenon of different correlation revealed in different test case regions can be explained by the effect under exceptional testing. The strong positive correlation in Region IV is caused by large number (277/373) of exceptional test cases contained in this region.

On the other hand, code coverage implies fault detection capability moderately in both functional testing and random testing. The difference between the two testing profiles is not obvious.

For the second question, we cannot give conclusive answer according to our data. The correlation pattern seems similar for all coverage metrics under various testing profiles. There is a small discrepancy between block coverage/C-use and decision coverage/P-use. It may be caused by the control flow diversion related to decision predicate. But as the difference is not statistically significant, we cannot tell whether the coverage metrics have influences on the concerned correlation.

As our project data are based on RSDIMU application, which is computation-intensive, the size of some functions is very large compared with other applications. We find that there is a gap between the coverage of different exceptional test cases, which is determined by the execution of these functions. It is the reason behind the two clusters shown in some of the patterns. As RSDIMU is a real-world application from critical avionics industry, the correlations and patterns that are observed in our experiment should be representative to a certain degree. However, since this is only a case study of investigation, further real-world empirical data are still needed.

The significance of the clear positive correlation in exceptional testing is that it can provide guidelines for selection and evaluation of exceptional test cases. Test cases with high code coverage tend to detect more faults, although it does not necessarily mean that test cases with low coverage are useless. For functional testing, test cases with low coverage may detect faults related to specified operations. For random testing or operational testing, code coverage can estimate the fault detection capability for exceptional test cases.

4.4 Summary

Software testing is a key procedure to ensure high quality and reliability of software programs. The key issue in software testing is the selection and evaluation of different test cases. Code coverage has been proposed to be an estimator for testing effectiveness, but it remains a controversial topic and lack of support from empirical data.

In this chapter, we employ coverage testing and mutation testing to investigate the relationship between code coverage and fault detection capability for test cases selection and evaluation purpose. A unique contribution *of our work is an innovative approach is establishing the relationship according to different testing profiles. We conduct a large-scale project with real-world application to address such relationship with different coverage metrics under different testing profiles. From our experimental data, code coverage is a moderate indicator for the capability of fault detection on the whole test set. The effect of code coverage on fault detection varies under different testing profiles. The correlation between the two measures is high with exceptional test cases, while weak in normal testing.*

Furthermore, there is no sign on various influence of different coverage metrics. All the four coverage metrics show similar patterns on the linear relationship between code coverage and fault detection. Moreover, the data support the effectiveness of random test cases due to its significant fault detection capability.

The new finding about the effect of code coverage on fault detection can be used to guide the selection and evaluation of test cases under various testing profiles, although this still needs supports and evaluations from more empirical data.

Chapter 5

Conclusion and Future Work

In this term paper, we first survey the background, techniques, reliability modeling and applications for software fault tolerance. Then based on our previous experiment on software reliability analysis on fault tolerance, we conduct further experiments and analyze the experimental data to learn the correlations among these faults and the relation to their resulting failures. we apply the experimental data on current famous reliability modeling to examine their effectiveness.

Furthermore, we investigate the effectiveness of data flow coverage and mutation coverage in testing for design diversity. We examine different hypotheses on software testing and fault tolerance schemes, and find that code coverage is a positive indicator for fault detection capability in software testing with our software fault tolerance project.

As our future work, we will focus on proposing our own fault correlation modeling to estimate the fault correlation among multiple versions/mutants. More investigations and experiments will be conducted based on statistical data. We are generating millions of random test cases and exercise them on the current versions to collect statistical fault correlation data.

Another possible research direction is the evaluation of test effectiveness. Although we have found that code coverage is a positive indicator for testing effectiveness, especially under the context of exceptional testing, a quantitative assessment about the relationship can be formulized.

Bibliography

[1] *T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding. Software fault tolerance: an evaluation.* IEEE Transactions on Software Engineering*, 12(1):1502–1510, January 1985.*

[2] *J. Arlat, K. Kanoun, and J. C. Laprie. Dependability modeling and evaluation of software fault-tolerant systems.* IEEE Transactions on Computers*, 39(4):504–513, September 1990.*

[3] *A. Avizienis. The n-version approach to fault-tolerant software.* IEEE Transactions on Software Engineering*, 11(12):1491–1501, December 1985.*

[4] *A. Avizienis. Dependable computing depends on structured fault tolerance. In* Proceedings of the 1995 6th International Symposium on Software Reliability Engineering*, pages 158–168, Toulouse, France, 1995.*

[5] *A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In* Proceedings of the Computer Software and Application Conference (COMPSAC77)*, pages 149–155, Chicago, Illinois, November 1977.*

[6] *A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. asic concepts and taxonomy of dependable and secure computing.* IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, December 2004.*

[7] *B. Beizer.* Software Testing Techniques. *Van Nostrande Reinhold Co., New York, 1990.*

[8] *F. Belli and P. Jedrzejowicz. Fault-tolerant programs and their reliability.* IEEE Transactions on Reliability*, 29(2):184–192, 1990.*

[9] *P. G. Bishop. Software fault tolerance by design diversity. In M. R. Lyu, editor,* Software Fault Tolerance*, pages 211–230. Wiley, New York, 1995.*

[10] *P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti. PODS - a project on diverse software.* IEEE Transactions on Software Reliability*, 12(9):929–940, 1986.*

[11] *R. J. Bleeg. Commercial jet transport fly-by-wire architecture considerations. In* AIAA/IEEE 8th Digital Avionics Systems Conference*, pages 399–406, October 1988.*

[12] *A. Bondavalli, S. Chiaradonna, F. D. Giandomenico, and L. Strigini. A contribution to the evaluation of the reliability of iterative-execution software.* Software Testing, Verification, and Reliability*, 9(3):145–166, March 1999.*

[13] *L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In* Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 124–157, West Palm Beach, Florida, November 1999.*

[14] *L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage.* IEEE Transactions on Reliability*, 49(1):60–70, March 2000.*

[15] *M. H. Chen, M. R. Lyu, and W. E. Wong. Effect of code coverage on software reliability measurement.* IEEE Transactions on Reliability*, 50(2):165–170, June 2001.*

[16] *M. H. Chen, A. P. Mathur, and V. J. Rego. Effect of testing techniques on software reliability estimates obtained using time domain models. In* Proceedings of the 10th annual software reliability symposium*, pages 116–123, Denver, Colorado, June 1992.*

[17] *X. Chen and M. R. Lyu. Message logging and recovery in wireless corba using access bridge. In* Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS2003)*, pages 107–114, Pisa, Italy, April 2003.*

[18] *F. Cristian. Exception handling and software fault tolerance. In* Proceedings of the 10th International Symposium on Fault-Tolerant Computing (FTCS-10)*, pages 97–103, 1980.*

[19] *F. Cristian. Exception handling and tolerance of software faults. In M. R. Lyu, editor,* Software Fault Tolerance*, pages 81–107. Wiley, New York, 1995.*

[20] *A. Csenki. Recovery block reliability analysis with failure clustering. In A. Avizienis and J. C. Laprie, editors,* Dependable Computing for Critical Applications(DCCA-1)*. Santa Barbara, USA, 1991.*

[21] *E. Delamaro, C. Maldonado, and A. Mathur. Interface mutation: An approach for integration testing.* IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.*

[22] *J. B. Dugan and M. R. Lyu. System reliability analysis of an N-version programming application.* IEEE Transactions on Reliability*, 43(4):513–519, December 1994.*

[23] *J. B. Dugan and M. R. Lyu. Dependability modeling for fault-tolerant software and systems. In M. R. Lyu, editor,* Software Fault Tolerance*, pages 109–138. Wiley, New York, 1995.*

[24] *D. E. Eckhardt, A. K. Caglavan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability.* IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.*

[25] *D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors.* IEEE Transactions on Software Engineering*, 11(12):1511–1517, December 1985.*

[26] *M. Ege, M. Eyler, and M. Karakas. Reliability analysis in n-version programming with dependent failures. In* Proceedings of 27th Euromicro Conference*, pages 174 –181, Warsaw, Poland, September 2001.*

[27] *R. C. et al. Orthogonal defect classification - a concept for in-process measurements.* IEEE Transactions on Software Engineering*, 18(19):943–956, November 1992.*

[28] *P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria.* IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.*

[29] *F. D. Frate, P. Garg, A. P. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In* Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 124–132, Toulouse, France, October 1995.*

[30] *A. Gnrarov, J. Arlat, and A. Avizienis. On the performance of software fault-tolerance strategies. In* Proceedings of 10th International Symposium on Fault Tolerant Computing (FTCS-10)*, pages 251–253, Kyoto,Japan, July 1980.*

[31] *J. B. Goodenough. Exception handling: issues and a proposed notation.* Communications of the ACM*, 18(12):683–693, 1975.*

[32] *K. E. Grosspietsch. Optimizing the reliability of the component-based n-version approaches. In* Proceedings of International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 138–145, Fort Lauderdale, Florida, April 2002.*

[33] *K. E. Grosspietsch and A. Romanovsky. An evolutionary and adaptive approach for n-version programming. In* Proceedings of 27th Euromicro Conference*, pages 182–189, Warsaw, Poland, September 2001.*

[34] *L. Hatton. N-version design versus one good version.* IEEE Software*, pages 71–76, Nov/Dec 1997.*

[35] *A. D. Hills and N. A. Mirza. Fault tolerant avionics. In* AIAA/IEEE 8th Digital Avionics Systems Conference*, pages 407–414, October 1988.*

[36] *J. R. Horgan, S. London, and M. R. Lyu. Achieving software quality with testing coverage measures.* IEEE Computer*, 27(9):60–69, September 1994.*

[37] *W. E. Howden. Weak mutation testing and completeness of test sets.* IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.*

[38] *Y. Huang and C. Kintala. Software fault tolerance in the application layer. In M. R. Lyu, editor,* Software Fault Tolerance*, pages 231–248. Wiley, New York, 1995.*

[39] *IEEE.* IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, New York, 1990.*

[40] *Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: a software infrastructure for adaptive fault tolerance.* IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.*

[41] *K. Kanoun. Real-world design diversity: a case study on cost.* IEEE Software*, pages 29–33, July/August 2001.*

[42] *J. Kelly, D. Eckhardt, M. Vouk, D. McAllister, and A. Caglayan. A large scale generation experiment in multi-version software: Description and early results. In* Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 9–14, June 1988.*

[43] *J. P. Kelly and A. Avizienis. A specification-oriented multi-version software experiment. In* Proceedings of the 13th Annual International Symposium on Fault-Tolerant Computing (FTCS-13)*, pages 120–126, Milano, June 1983.*

[44] *K. Kim, M. A. Vouk, and D. F. McAllister. An empirical evaluation of maximum likelihood voting in high inter-version failure correlation conditions. In* Proceedings of the 7th International Symposium on Software Reliability Engineering*, pages 330–339, October 1996.*

[45] *K. H. Kim. Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults. In* Proceedings of the 4th International Conference on Distributed Computing Systems, *pages 526–532, 1984.*

[46] *K. H. Kim. The distributed recovery block scheme. In M. R. Lyu, editor,* Software Fault Tolerance, *pages 189–210. Wiley, New York, 1995.*

[47] *J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming.* IEEE Transactions on Software Engineering, *12(1):96–109, January 1986.*

[48] *J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures.* IEEE Computer, *23(7):39–51, July 1990.*

[49] *J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Architectural issues in software fault tolerance. In M. R. Lyu, editor,* Software Fault Tolerance, *pages 47–80. Wiley, New York, 1995.*

[50] *J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun, and C. Hourtolle. Hardware and software fault tolerance: definition and analysis of architectural solutions. In* Proceedings of the 17th International Symposium on Fault-Tolerant Computing (FTCS-17), *pages 116–121, Pittsburgh, PA, 1987.*

[51] *J. C. Laprie and K. Kanoun. Software reliability and system reliability. In M. R. Lyu, editor,* Handbook of Software Reliaiblity Engineering, *pages 27–69. McGraw-Hills, New York, 1996.*

[52] *P. A. Lee and T. Anderson.* Fault Tolerance: Principles and Practice. *Springer-Verlag, New York, 1990.*

[53] *B. Littlewood and D. Miller. Conceptual modeling of coincident failures in multiversion software.* IEEE Transactions on Software Engineering, *15(12):1596–1614, December 1989.*

[54] *B. Littlewood, P. Popov, and L. Strigini. Design diversity: an update from research on reliability modelling. In* Proceedings of the 21th Safety-Critical Systems Symposium, *Bristol, U.K., 2001.*

[55] *B. Littlewood, P. Popov, and L. Strigini. Modelling software design diversity - a review.* ACM Computing Surveys, *33(2):177–208, June 2001.*

[56] *B. Littlewood, P. Popov, and L. Strigini. Assessing the reliability of diverse fault-tolerant software-based systems.* Safety Science, *40:781–796, 2002.*

[57] *M. R. Lyu.* A Design Paradigm for Multi-Version Software. *PhD thesis, UCLA, Los Angeles, May 1988.*

[58] *M. R. Lyu, editor.* Software Fault Tolerance. *Wiley, New York, 1995.*

[59] *M. R. Lyu, editor.* Handbook of Software Reliability Engineering. *McGraw-Hill and IEEE Computer Society, New York, 1996.*

[60] *M. R. Lyu. Reliability-oriented software engineering: Design, testing, and evaluation techniques.* IEE Software Proceedings, *145(6):191–197, December 1998.*

[61] *M. R. Lyu and Y. T. He. Improving the N-version programming process through the evolution of a design paradigm.* IEEE Transactions on Reliability, *42(2):179–189, March 1993.*

[62] *M. R. Lyu, J. R. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing. In* Proceedings of the 4th International Symposium on Software Reliability Engineering, *pages 25–34, Denver, November 1993.*

[63] *M. R. Lyu, J. R. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing.* IEEE Transactions on Reliability, *43(4):527–535, December 1994.*

[64] *M. R. Lyu, Z. Huang, K. S. Sze, and X. Cai. An empirical study on testing and fault tolerance for software reliability engineering. In* Proceedings 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003), *pages 119–130, Denver, Colorado, November 2003.*

[65] *Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich. Software reliability growth with test coverage.* IEEE Transactions on Reliability, *51(4):420–426, December 2002.*

[66] *J. Musa.* Software Reliability Engineering: More Reliable Software Faster and Cheaper. *AuthorHouse, 2nd edition, 2004.*

[67] *P. G. Neuman.* Computer Related Risks. *Addison-Wesley, Boston, 1995.*

[68] *V. F. Nicola. Checkpointing and the modeling of program execution time. In M. R. Lyu, editor,* Software Fault Tolerance, *pages 167–188. Wiley, New York, 1995.*

[69] *A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators.* ACM Transactions on Software Engineering Methodology, *5(2):99–118, 1996.*

[70] *A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths.* The Journal of Software Testing, Verification, and Reliability, *7(3):165–192, September 1997.*

[71] *J. Offutt and S. D. Lee. An empirical evaluation of weak mutation.* IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.*

[72] *P. Popov and L. Strigini. Diversity with off-the-shelf components: a study with SQL database servers. In* Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, pages B84–B85, 2003.*

[73] *P. T. Popov, L. Strigini, J. May, and S. Kuball. Estimating bounds on the reliability of diverse systems.* IEEE Transactions on Software Engineering*, 29(4):345–359, April 2003.*

[74] *D. K. Pradhan.* Fault Tolerant Computer System Design*. Prentice Hall, New Jersey, 1996.*

[75] *L. L. Pullum.* Software Fault Tolerance Techniques and Implementation*. Artech House, Boston, 2001.*

[76] *B. Randell. System structure for software fault tolerance.* IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.*

[77] *B. Randell and J. Xu. The evolution of the recovery block concept. In M. R. Lyu, editor,* Software Fault Tolerance*, pages 1–21. Wiley, New York, 1995.*

[78] *S. Rapps and E. J. Weyuker. Selecting software test data using data flow information.* IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.*

[79] *R. K. Scott, J. W. Gault, and D. F. McAllister. Fault tolerant software reliability modeling.* IEEE Transactions on Software Engineering*, 13(5):582–592, 1987.*

[80] *L. Strigini. On testing process control software for reliability assessment: the effects of correlation between successive failures.* Software Testing Verification and Reliability*, 6(1):33–48, January 1996.*

[81] *S. K. Sze and M. R. Lyu. ATACOBOL: A COBOL test coverage analysis tool and its applications. In* Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'2000)*, pages 327–335, San Jose, California, October 2000.*

[82] *A. T. Tai, A. Avizienis, and J. F. Meyer. Performability enhancement of fault-tolerant software.* IEEE Transactions on Reliability, Special Issue on Fault-Tolerant Software*, 42(2):227–237, June 1993.*

[83] *X. Teng and H. Pham. A software-reliability growth model for n-version programming systems.* IEEE Transactions on Reliability*, 51(3):311–321, September 2002.*

[84] *L. A. Tomek and K. S. Trivedi. Analyses using stochastic reward nets. In M. R. Lyu, editor,* Software Fault Tolerance*, pages 139–165. Wiley, New York, 1995.*

[85] *W. Torres-Pomales. Software fault tolerance: a tutorial. Technical Report TM-2000-210616, NASA Langley Research Center, Hampton, Virginia, Oct. 2000.*

[86] *P. Townend and J. Xu. Fault tolerance within a grid environment. In* Proceedings of the UK e-Science All Hands Meeting 2003*, pages 272–275, Nottingham, UK, September 2003.*

[87] *P. Traverse. Dependability of digital computers on board airplanes. In* Proceedings of the 2nd IFIP Working Conference on Dependable Computing for Critical Applications*, pages 133–152, Tucson, Arizona, 1991.*

[88] *M. A. Vouk, A. Caglayan, D. E. Eckhardt, J. Kelly, J. Knight, D. McAllister, and L. Walker. Analysis of faults detected in a large-scale multi-version software development experiment. In* Proceedings of Digital Avionics Systems Conference*, pages 378–385, October 1990.*

[89] *E. J. Weyuker. The cost of data flow testing: an empirical study.* IEEE Transactions on Software Engineering*, 16(2):121–128, February 1988.*

[90] *T. W. Williams, M. R. Mercer, J. P. Mucha, and R. Kapur. Code coverage: what does it mean in terms of quality? In* Proceedings of the Annual Reliability and maintainability Symposium*, pages 420–424, Philadelphia, PA, USA, January 2001.*

[91] *W. Wong and A. Mathur. Reducing the cost of mutation testing: An empirical study.* The Journal of Systems and Software*, 31(3):185–196, December 1995.*

[92] *W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In* Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 230–238, Monterey, CA, November 1994.*

[93] *J. Xu and B. Randell. Software fault tolerance: t/(n-1)-variant programming.* IEEE Transactions on Reliability*, 46(1):60–68, March 1997.*