Coverage-Based Testing Strategies and Reliability Modeling for Fault-Tolerant Software Systems

CAI Xia

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

September 2006

Coverage-Based Testing Strategies and Reliability Modeling for Fault-Tolerant Software Systems

submitted by

CAI Xia

for the degree of Doctor of Philosophy

at the Chinese University of Hong Kong

Abstract

Software permeates our modern society, and its complexity and criticality is ever increasing. Thus the capability to tolerate software faults, particularly for critical applications, is evident. While fault-tolerant software is seen as a necessity, it also remains as a controversial technique and there is a lack of conclusive assessment about its effectiveness.

This thesis aims at providing a quantitative assessment scheme for a comprehensive evaluation of fault-tolerant software including reliability model comparisons and trade-off studies with software testing techniques. First of all, we propose a comprehensive procedure in assessing fault-tolerant software for software reliability engineering, which is composed of four tasks: modeling, experimentation, evaluation and economics. Our ultimate objective is to construct a systematic approach to predicting the achievable reliability based on the software architecture and testing evidences, through an investigation of testing and modeling techniques for fault-tolerant software.

Motivated by the lack of real-world project data for investigation on software testing and fault tolerance techniques together, we conduct a real-world project and engage multiple programming teams to independently develop program versions based on an industry-scale avionics application. Detailed experimentations are conducted to study the nature, source, type, detectability, and effect of faults uncovered in the program versions, and to learn the relationship among these faults and the correlation of their resulting failures. Coverage-based testing as well as mutation testing techniques are adopted to reproduce mutants with *real* faults, which facilitate the investigation on the effectiveness of data flow coverage, mutation coverage, and fault coverage for design diversity.

Then based on the preliminary experimental data, further experimentation and detailed analyses on the correlations among these faults and the relation to their resulting failures are studied. The results are further applied to the current reliability modeling techniques for fault-tolerant software to examine their effectiveness and accuracy.

Furthermore, to investigate some “variants” as well as “invariants” of fault-tolerant software, we perform an empirical investigation on evaluating reliability features by a comprehensive comparison between two projects: our project and NASA 4-University project. Based on the same specification for program development, these two projects encounter some common as well as different features. The testing results of two comprehensive operational testing procedures involving hundreds of thousands test cases are collected and compared. Similar as well as dissimilar faults are observed and analyzed, indicating common problems related to the same application in both projects. The small number of coincident failures in the two projects, nevertheless, provide a supportive evidence for N-version programming, while the observed reliability improvement implies some trends in the software development in the past twenty years.

Next, we investigate the effect of code coverage on fault detection which is the underlying intuition of coverage-based testing strategies. From our experimental data, we find that code coverage is a moderate indicator for the capability of fault detection on the whole test set. But 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, but weak in normal testing. Moreover, our study shows that code coverage can be used as a good filter to reduce the size of the effective test set, although it is more evident for exceptional test cases.

Finally, we formulate the relationship between code coverage and fault detection. Although our two current models are in simple mathematical formats, they can predict the percentage of fault detected by the code coverage achieved for a certain test set. We further incorporate such formulation into traditional reliability growth models, not only for fault-tolerant software, but also for general software system. Our empirical evaluations show that our new reliability model can achieve more accurate reliability assessment than the traditional Non-homogenous Poisson model.

容錯軟體系統的覆蓋測試策略與可靠性模型

論文摘要

本論文討論容錯軟體系統的基於代碼覆蓋率的測試策略與可靠性模型。

為了給全面地評測容錯軟體，包括可靠性模型的比較，軟體測試與軟體容錯之間的平衡，提供一個可度量的評價體系，我們首先提出了一套完整的研究方法與步驟來測量容錯軟體系統的軟體可靠性工程，包括四個過程：建模，實驗，評測與平衡。我們的最終目標是通過對容錯軟體測試與可靠性度量技術的學習，構建一個系統化的、根據軟體結構與測試資料來預測容錯軟體系統可達到的可靠性水準的方法。

首先，基於目前實驗與測試資料的缺乏，我們設計了一個航空應用的程式專案，最終得到由多個開發小組獨立完成的多組程式。在此基礎上，我們調查了這多組程式在開發過程中出現錯陷的來源、特點、類型、可測試性與效果，並研究了這些錯陷與軟體失敗間的關係。我們採用了變異測試技術以生成融入真實缺陷的變異體，同時也採用了覆蓋測試法來採集資料流程覆蓋率，變異體覆蓋率及缺陷查找率等數據。

在這些程式資料的基礎上，我們進一步分析了軟體缺陷與軟體失敗之間的關聯，並將其應用於目前存在的基於容錯軟體的可靠性模型上，以檢測其有效性與準確度。

同時，為了分析容錯軟體系統的一些“變數”與“非變數”，我們將自己的實驗資料與ＮＡＳＡ資助的美國四所大學于１９８７年聯合完成的實驗資料做了完整地比較與研究。這兩個實驗中同時出現於多個版本之間的軟體失敗數很少，驗證了多版本程式的有效性。同時兩個實驗的對比也顯示了軟體發展在過去二十年的發展趨勢。

另外，我們還著重研究了代碼覆蓋率對軟體缺陷檢測數目的影響。我們發現在整個測試集上，代碼覆蓋率能適當地預測測試用例對軟體缺陷的檢測能力。但是，這種預測作用在不同的測試策略上是不同的，例如，在異常測試用例時很高，而在正常測試用例時很低。另外，我們的分析顯示代碼覆蓋率是一個很好的用來減小測試集數目的指標，儘管它對異常測試用例更有效。

最後，我們對代碼覆蓋率與缺陷檢測率之間的關係建模。儘管這兩個模型只描述了兩者之間的簡單的數學關係，它們卻能較準確地預測兩者之間的關係。這種覆蓋率之間的關係還能被融合進傳統的基於測試時間的軟體可靠性模型，以提高容錯軟體系統的可靠性預測。

Acknowledgment

I would like to take this opportunity to express my sincere gratitude to my supervisor, Prof. Michael R. Lyu. My Ph.D study would never have been completed without his exceptional guidance and consistent support. I have learned a lot from his breadth of knowledge, his enthusiasm for research, as well as his patience and encouragement. His inspiring advice are extremely essential and valuable in this research work.

I am so appreciated for all the support and help from Prof. Mladen A. Vouk. He has spent a lot of his precious time to dig out the NASA 4-University project data for us, and has given us many valuable comments for the comparison of the two projects. Many thanks go to Prof. Lorenzo Strigini. During his short visiting at CUHK, he gave us many inspired suggestions and insights for possible research directions.

I am so grateful to Prof. Ada Fu, Prof. Jeffrey Yu and Prof. Shing-Chi Cheung for their precious time to serve as my thesis committee, as well as their valuable comments and feedback on this work.

I would like to thank Dr. Haixuan Yang for his valuable discussions about the mathematical formulations. Many thanks go to Dr. Xinyu Chen, Prof. Ping Guo, Mr. Jianke Zhu and Ms. Huiye Ma, for their help, encouragement and discussions with this research work.

Last but not least, I would like to thank my husband Xuetai Zhang, for his love, patience, encouragement, and support. I also express my appreciation to my parents. They not only teach me the meaning of life, the importance of study, and the significance of love; but also always stand behind me when I need love, support and help. Particularly, I want to thank my four-year-old daughter Shitao. Her love, patience and prayers encourage me a lot when I finalize this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Having attracted major attentions from academia as well as industry, software reliability engineering techniques can be classified in the following areas: fault avoidance, fault removal, fault tolerance, and fault prediction [70]. Traditionally, software reliability is achieved by fault avoidance techniques (including structure programming, software reuse, and formal methods) to prevent software faults, or by fault removal techniques (including testing, verification, and validation) to detect and eliminate software faults. As the complexity of software increases, the number of dormant software faults present at system operation also increases. Therefore, the capability to tolerate software faults, particularly for critical applications, is evident. These software faults may or may not be manifested during system operations, but if they do, fault tolerant software techniques should provide the necessary mechanisms to prevent system failure from occurring. Finally, if some software faults or failures cannot be avoided, removed or tolerated, they are expected to be predictable at least. Thus fault/failure prediction has been the main focus of software reliability modeling, which tries to predict the failure behavior under operational profile on the basis of the failure data collected in testing phases.

In the area of software fault tolerance, on the one hand, fault tolerant software is seen as a necessity when the complexity of software increases rapidly;

on the other hand, it also remains as a controversial technique and there is a lack of conclusive assessment about its effectiveness. As one of the main techniques for software fault tolerance, design diversity 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 [69]. 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. Because of these, although several probability reliability models have been proposed to estimate the overall reliability under fault correlation assumptions, more investigations and evaluations are needed. Furthermore, little research work has been engaged in answering the questions such as how to test for fault tolerance, and how effective fault tolerant software can achieve.

The purpose of this research is to assess fault tolerant software for software reliability engineering. In this thesis, we perform systematic investigation and evaluation the performance of software fault tolerance techniques. Particularly, we focus our study on testing strategies and reliability modeling for fault tolerant software.

In the following, we will introduce current testing strategies in general and reliability models for fault-tolerant software briefly. Based on that, we will highlight the main contributions as well as the organizations of this thesis.

1.1 The Effectiveness of Software Testing Strategies

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.

According to the test case design principle, there are two major testing schemes, they are: subdomain-based testing and random testing. *Subdomain-based testing* inherit the feature that the input domain is divided into subsets, called subdomains, and one or more representatives from each subdomain are selected to form the final test set [33]. This approach is also referred partition testing if the subdomains are independent [40, 105]. In contrast to subdomain-based testing, *random testing* simply generates test cases within the entire input domain [26]. With random testing, it is easier to design large numbers of test cases to perform quantitative reliability analysis of programs.

Within subdomain-based testing, there are *Functional testing* (so-called *black-box testing*) and *Structural testing* (so-called *white-box testing*). *Mutation testing* is also known as one of subdomain-based testing, although it begins by creating many "faulty" versions of a program [47].

For various testing strategies, the effectiveness and completeness of the test sets has remained an active research issue over the past several decades. The comparison of functional testing, structural testing, and random testing has also drawn a great deal of research interest.

Furthermore, for the purpose of test case selection and evaluation, code coverage has been proposed as an indicator of testing effectiveness and completeness in order to improve the test resource allocation [73, 91, 96]. However, it remains a controversial issue about whether code coverage is a good indicator for fault detection capability of test cases. Some empirical studies have shown that high code coverage brings high software reliability and low fault rate [30, 45, 91, 104]. It is also observed 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 [34].

On the other hand, despite the observations of correlation existing in code coverage and fault coverage, a question is raised: 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 in [13] 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.

1.2 Reliability Modeling for Fault-Tolerant Software

On the fault tolerance side, the main technique is software design diversity, including recovery blocks [89], N-version programming [4], and N self-checking programming [61]. Design diversity approach achieves fault-tolerant software systems through the independent development of program versions from a common specification.

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 [28] proposed the first model of fault correlation for diverse systems. Later Littlewood and Miller [64] 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 [25] proposed a dependability model for N-version programming to parameterize the possibility of fault correlations. Recently, Popov Strigini et al [86] 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.

Among these proposed reliability models, some of them are too theoretical that they cannot be evaluated in real projects, others are based on strong assumptions which are not true in practice. Besides, as fault-tolerate software costs much more than one version software, the empirical data are rarely available. In summary, on the reliability models for fault-tolerant software, theoretical yet practical reliability model, as well as suitable empirical data with real-world fault-tolerant software systems, are highly demanded.

1.3 Contributions of this Thesis

In this thesis, we perform thorough investigations and evaluations on coverage-based testing strategies and reliability modeling for fault-tolerant software systems. The major contributions are listed as follows:

- Formulate the relationship between fault detected and test coverage achieved in testing phase; based on the new formulations, a new reliability model is proposed which incorporates the code coverage measurement on the traditional time-based reliability models.
- Assess the effect of code coverage on fault detection in various testing strategies; our findings support that code coverage is clearly a good indicator for fault detection capability in exceptional test cases.
- Evaluate current famous reliability models for fault-tolerant software and compare their effectiveness and accuracy.
- Cross-compare two major fault-tolerant software projects and assess the "variants" as well as "invariants" features for design diversity.
- Conduct a large-scale multi-version project with real-world application, perform thorough acceptance and operational testing, generate hundreds of mutants, and collect valuable faults/failures data for further investigations and evaluations on software reliability, software testing and software fault tolerance.

1.4 Organization of this Thesis

The remainder of this thesis is organized as follows. A detailed background study on fault-tolerant software, its reliability models and software testing is performed in the next chapter. In Chapter 3, we illustrate our research methodology which runs through this thesis. The setup and preliminary data of the large-scale multi-version software project is thoroughly described in Chapter 4. Next, in Chapter 5, two reliability models for fault-tolerant software are evaluated and compared using our experimental data. Qualitative as well as quantitative comparisons between two design diversity experiments are conducted in Chapter 6. Then Chapter 7 evaluates the effect of code coverage on fault detection under various testing strategies and coverage measurements. A novel reliability model is formulated on the basis of code coverage as an indicator for fault detection in Chapter 8. Finally, Chapter 9 summarizes this thesis and illustrates our future work.

□ **End of chapter.**

Chapter 2

Background and Related Work

In this Chapter, we perform the background study in the two main topics related to this research effort: fault-tolerant software and its reliability models, and software testing strategies. We are motivated by this survey and related work to focus our work on the potential relationship between software fault tolerance, software testing and software reliability modeling.

2.1 Fault-Tolerant Software and its Reliability Models

Fault tolerance is the survival attribute of a system or component to continue operating as required despite the manifestation of hardware or software faults [49]. 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 [69]. When a fault occurs, fault-tolerant software provides mechanisms to prevent the system failure from occurring [88].

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*.

2.1.1 Historical Background

Most of the fault-tolerant software techniques were introduced and proposed in 1970s. For example, as one of single version fault-tolerant software techniques, the exception handling approach began to appear in the 1970s, and a wide range of investigations in this approach led to more mature definitions, terminology and exception mechanisms later on [22]. Another technique, checkpointing and recovery, was also commonly employed to enhance software reliability with efficient strategies [80].

In the early 1970s, a research project was conducted at the University of Newcastle [90]. The idea of the recovery block (RB) evolved from this project and 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 [4], which involved redundancy of three basic elements in the approach: process, product and environment [3]. N self-checking programming approach was introduced most recently, yet it was based on the concept of self-checking programming which had long been introduced [60].

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

Definitions

As fault-tolerant software is capable of providing the expected service despite the presence of software faults [4, 89], we first introduce the concepts related to this technique [62].

Failures. A failure occurs when the user perceives that a software program is unable to deliver the expected service [60]. 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 a 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 a previously saved state and executed again.

Fault-tolerant software thus attempts to prevent failures by tolerating software errors caused by software faults, particularly design faults. The progression "fault-error-failure" shows their causal relationship in a software lifecycle, as illustrated in Figure 2.1. Consequently, there are two major groups of approaches to deal with design faults: 1) fault avoidance (prevention) and fault

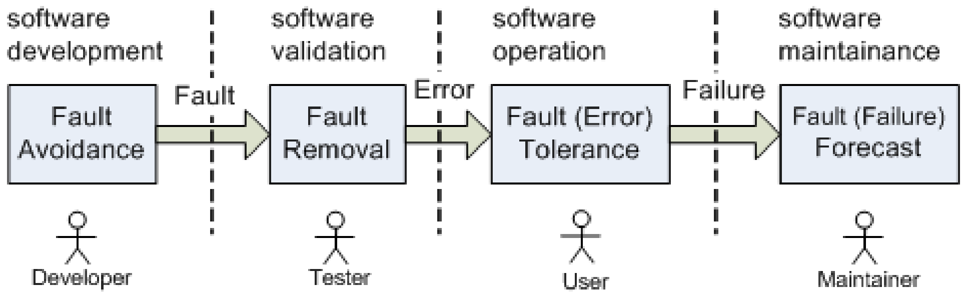

Figure 2.1: The transition of fault, error and failure in a software lifecycle

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 prediction (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.

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 incompleteness involved in software design, implementation and testing phases. The aim of fault-tolerant software, thus, is to prevent software faults from resulting in incorrect operations, including severe situations such as hanging or as the worst, crashing the 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.

Practice

From a user's point of view, fault tolerance represents two dimensions: availability and data consistency of the application [48]. Generally, there are four layers of fault tolerance. The top layer is composed of general fault tolerance techniques which are applicable to all applications, including checkpointing, exception handling, recovery block, N-version programming, N-self checking programming and other approaches. Some of the top-level techniques will be

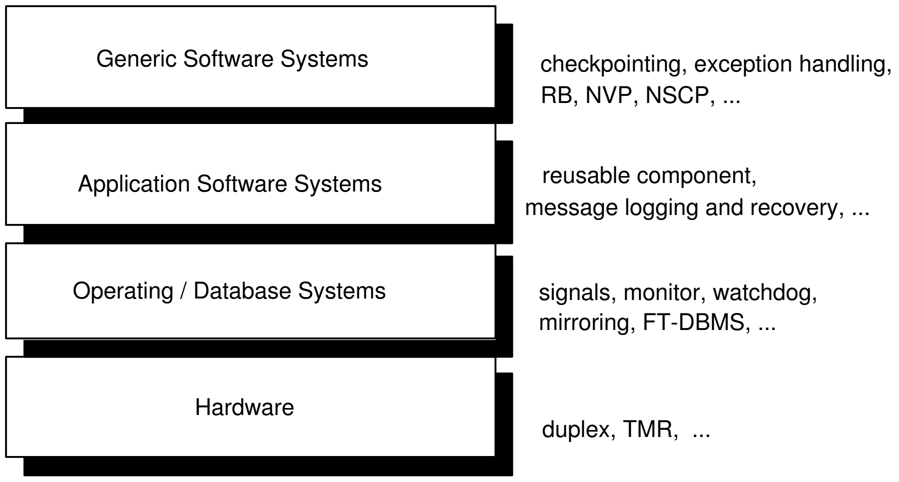
Generic Software Systems
checkpointing, exception handling, RB, NVP, NSCP, ...
Application Software Systems
reusable component, message logging and recovery, ...
Operating / Database Systems
signals, monitor, watchdog, mirroring, FT-DBMS, ...
Hardware
duplex, TMR, ...

Figure 2.2: Layers of fault tolerance

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. These are traditional hardware fault-tolerant techniques including duplex, triple modular redundancy (TMR), symmetric multiprocessing (SMP), shared memory and so on. Summary of these 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 /A380 aircraft) [11, 44, 75], aerospace applications [79], nuclear reactors, telecommunications products [48], network systems [57], and other critical software systems.

2.1.2 Fault-Tolerant Software Techniques

We examine two different groups of techniques for fault-tolerant software: single version and multi-version software techniques [69]. *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* [69, 99].

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

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

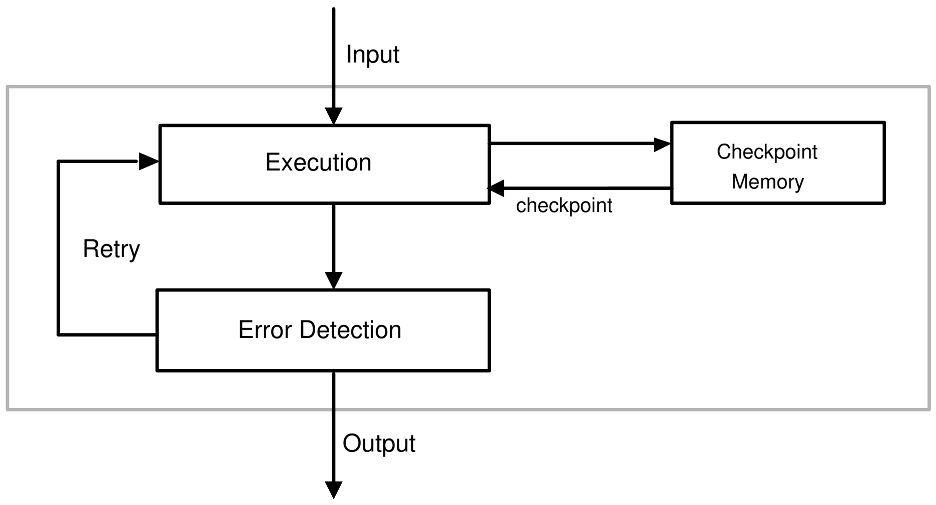

Figure 2.3: Logic of checkpoint and recovery

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 [87]. *Checkpointing* is used in (typically backward) error recovery, by saving the state of a system periodically. When an error 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 execution.

There are two kinds of checkpointing and recovery schemes: single process systems with a single node, and multiple communicating processes on multiple nodes [88]. 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 [88].

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 [63]. The structure of exception handling is shown in Figure 2.4.

Exception handling, proposed in the 1970's [37], is often considered as a limited approach to fault-tolerant software [21]. 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.

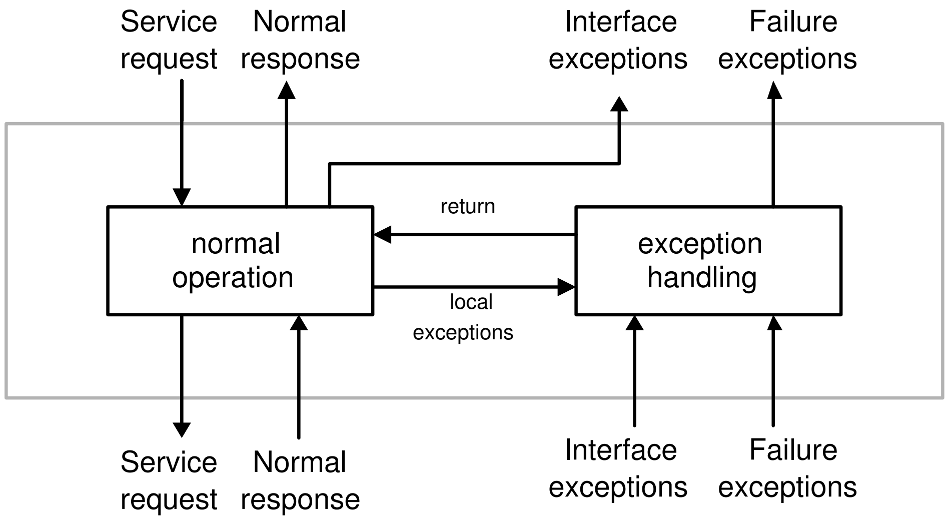

Figure 2.4: Logic of exception handling

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.

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 [4]. 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 [89, 90]. 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 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

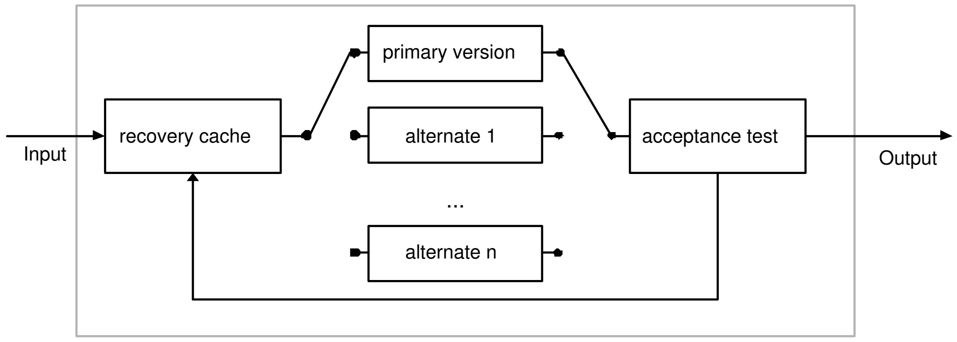

Figure 2.5: The recovery block (RB) model

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.

N-Version Programming

The concept of *N-version programming* (NVP) was first introduced in 1977 [4]. 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 [69]. 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

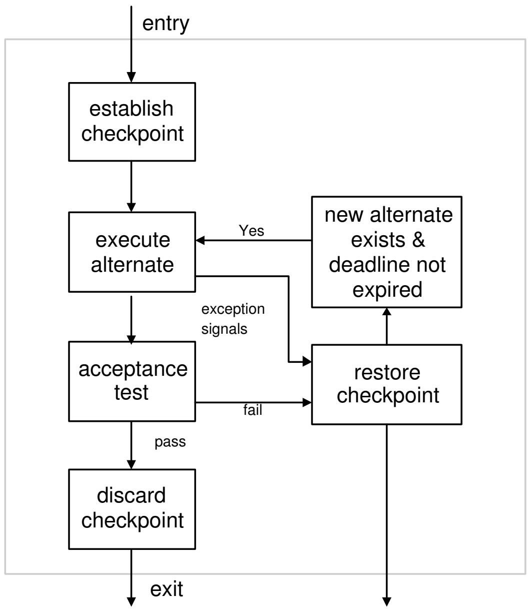

Figure 2.6: Operation of recovery block

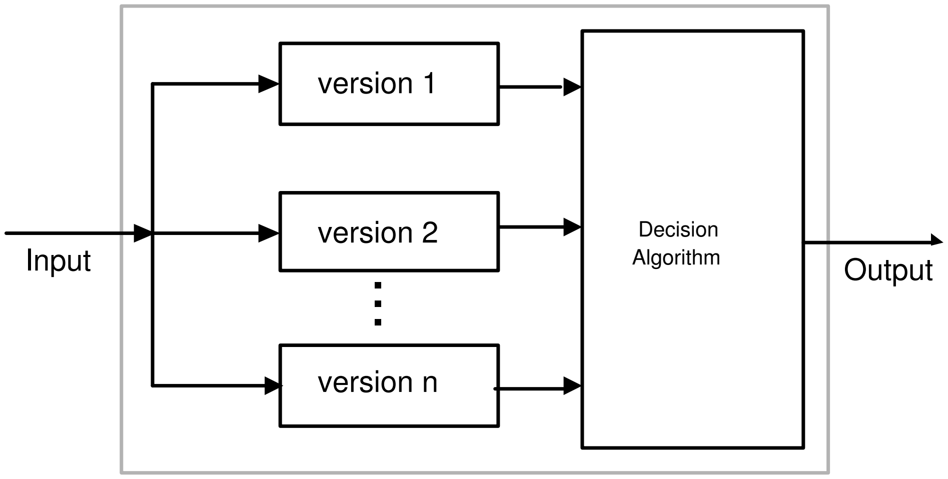

Figure 2.7: The N-version programming (NVP) model

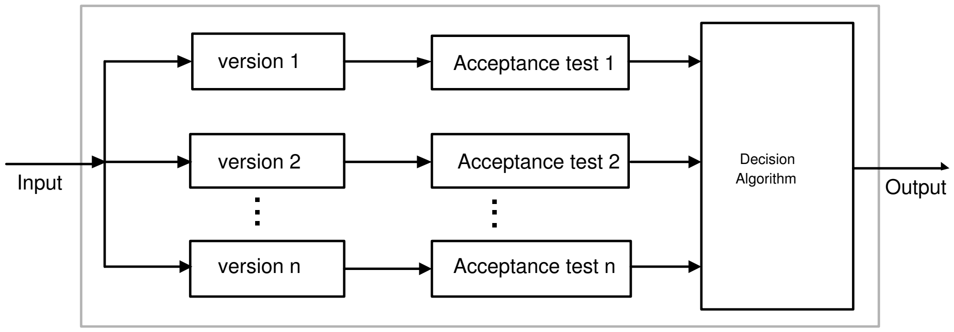

Figure 2.8: N self-checking programming using acceptance test

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 [3, 9, 27, 58].

N-Self Checking Programming

N self-checking programming (NSCP) was developed in 1987 by Laprie et al. [60, 61]. It involves the use of multiple software versions combined with structural variations of the recovery block and N-version programming approaches. Both acceptance tests 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 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

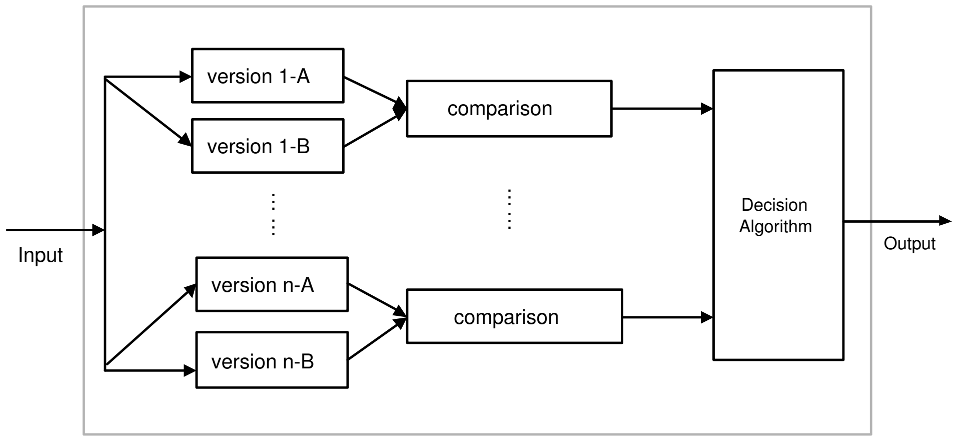

Figure 2.9: N self-checking programming using decision algorithm

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 difficult in implementation, but it can still

Table 2.1: Comparison of design diversity techniques

Features	recovery block	N-version programming	N self-checking programming
Minimum no. 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

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 [56], 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 [94]. 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 [110]. 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.

2.1.3 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 [88]. 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.

Eckhardt and Lee's Model

Eckhardt and Lee (EL Model) [28] 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.

Littlewood and Miller's Model

Littlewood and Miller [64] (LM model) showed that the variation of difficulty could be turned from a disadvantage into a benefit with forced design diversity [86]. "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.

Dugan and Lyu's Dependability Model

The dependability model proposed by Dugan and Lyu in [25] 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 [25], 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.

Tomek and Trivedi's Stochastic Reward Nets Model

Stochastic reward nets (SRNs) are a variant of stochastic Petri nets. SRNs are employed in [98] 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.

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 [86]. 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.

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 [54, 69], NASA 4-University project [27, 86, 103], Knight and Leveson's experiment [58], Lyu-He study [25, 71], 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, 4, 10, 42, 53, 55, 58]; 2) experiments evaluating the design process of diverse systems [3]; and 3) adoption of design diversity into different aspects of software engineering practice [71, 74].

To investigate the effectiveness of design diversity, an early experiment [4], 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 [58] 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. [55] 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 [42] 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 [53] 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 [4], a six language NVP project was conducted using a proposed *N-version Software Design Paradigm*

[68]. 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 [103] 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 [27, 64, 86] to validate the effectiveness of different modeling schemes.

The Lyu-He study [71] was derived from an experimental implementation involving 15 student teams guided by the evolving NVP design paradigm in [3]. 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 [25] to evaluate the prediction accuracy of Dugan and Lyu's Model. Lyu et al [74] 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.

2.1.4 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 [44] and AIRBUS A320/A330/A340/A380 aircraft [75, 101]), aerospace applications [79], nuclear reactors, telecommunications products [48], network systems [57], 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 [85].

We may summarize the fault-tolerant software applications into four categories: 1) reusable component library, e.g., [48]; 2) checkpointing and recovery schemes, e.g., [19, 87]; 3) entity replication and redundancy, e.g., [52, 100]; 4) early applications and projects on design diversity, e.g., [44, 85, 101]. An overview of some of these applications is given below.

Huang and Kintala [48] 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 [87], 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 [19] 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 [100] 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 [52] 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 [44] and AIRBUS A320/A330/A340/A380 aircraft [75, 101]. 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 [101], 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 [85] 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.

2.2 Software Testing Strategies

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 design and evaluation. Exhaustive testing, which test all possible inputs, is generally not applicable as most of the input domain is very large, even infinite. Thus various testing strategies have been proposed to design an effective test set in order to detect as many faults as possible.

2.2.1 Current Testing Strategies

Among these strategies, *Functional testing* (so-called *black-box testing*) aims at testing functions which are developed based on either the requirement or the specification [46], e.g., *Specification-based testing* [43].

Structural testing (so-called *white-box testing*) requires certain parts of the program code to be executed by the test set [51]. For example, *branch testing* requires every program branch to be executed at least once in one test set, while *data-flow coverage testing* [91] measure the test completeness by executing the test cases and measuring how definition and usage of certain types of variables are exercised.

Unlike the two testing strategies above, *mutation testing* begins by creating many "faulty" versions of a program [47]. The effectiveness of a test case is evaluated by whether it can cause each faulty version to fail.

Almost all these testing strategies inherit a common feature: the input domain is divided into subsets, called *subdomains*, and one or more representatives from each subdomain are selected to form the final test set. This approach is called *partition testing* [40, 105]. It has more recently been argued that some of the input subdomains may not be disjoint but overlapping [33], so these testing strategies were referred as *subdomain-based testing.*

In contrast to subdomain-based testing, *random testing* simply generates test cases within the entire input domain [26]. With random testing, it is easier to design large numbers of test cases to perform quantitative reliability analysis of programs.

In terms of the ability to detect faults, various testing strategies have been evaluated and compared through experiments [32], simulations [26, 40], and analysis [12, 18, 31, 33, 50, 78, 105]. Furthermore, based on the intuition that more faults will be revealed if more code is executed during testing, code coverage has been proposed as an indicator of testing effectiveness and completeness for the purpose of test case selection and evaluation [73, 91, 96]. However, as this remains a controversial issue, more empirical test data with real-world complicated applications are seriously needed to evaluate the effect of code coverage on test case evaluation and selection under various testing strategies.

For various testing strategies, the effectiveness and completeness of the test sets has remained an active research issue over the past several decades. For structural testing, code coverage is supposed to be an indicator of the fault detection capability in a given test case. Moreover, the comparison of functional testing, structural testing, and random testing has also drawn a great deal of research interest.

2.2.2 Code Coverage: Definition and Indication

Structural testing is based on the intuition that more faults will be revealed if more code is executed during testing. Based on this intuition, code coverage has been proposed as an indicator of testing effectiveness and completeness for the purpose of test case selection and evaluation [73, 91, 96]. Code coverage is measured as the fraction of program code that is executed at least once during the test. Various code coverage criteria have been suggested [45], including block coverage, decision coverage, C-use coverage and P-use coverage.

The definitions of these different coverage criteria are given in [45]. We give brief descriptions of each as follows:

Block coverage is measured as the portion of basic blocks executed. Basic blocks are maximal code fragments without branching, which contains no internal control flow 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.

However, it remains a controversial issue whether code coverage is a good indicator for fault detection capability of test cases. Some previous studies have shown that high code coverage brings high software reliability and low fault rate [30, 45, 91, 104]. Such experimental data indicate that both code coverage and number of faults detected in programs grow over time, as testing progresses. For example, [17] observed this correlation between code coverage and software reliability using experimentation with randomly generated flow graphs. In [107], it was reported that the correlation between test effectiveness and block coverage is higher than that between test effectiveness and size of test set. [34] 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 as a positive indicator for software reliability and quality, some researchers have tried to model the relationship between code coverage and code quality by hypergeometric distribution modeling [106] (under the assumption of a uniform probability and a random distribution of defects in the unit code, and independence between defects). Some have suggested code coverage as an additional parameter for the prediction of software failures in operation [15]. Others have modeled the relation among testing time, coverage and reliability [76].

On the other hand, despite these observations of correlations between code coverage and fault coverage, a question is raised [13]: 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 with 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.

2.2.3 Comparisons of Different Testing Strategies

Considerable research attention has been paid to the comparison of the effectiveness of partition testing versus random testing [12, 18, 26, 33, 40, 81, 105]. Duran and Ntafos [26] began the first comparison through a simulation study. This showed that the performance of random testing is very close to that of partition testing, yet the former may be more cost-effective than the latter. [40] performed a more extensive simulation for further investigation and showed similar results.

More recently, several analytical studies have been conducted to ascertain sufficient or necessary conditions under which one strategy performs better than the other [12, 18, 33, 78, 81, 105]. Particularly, [105] states that, under the condition that all the partitions have the same size and the same number of test cases are selected from each subdomain, partition testing has equal or better performance. This condition was generalized in [18], by stating that partition testing has at least equal performance to random testing if the allocation of the test cases to the subdomains is proportional to the size of the subdomains. This concept of *proportional partition testing* was challenged by [81] through simulations. However, all these studies used synthetic data to illustrate their analytical results. It has also been argued that the factors that affect the performance between partition testing and random testing need to be studied through data from real projects.

Formal analysis of the fault detection ability of various testing methods was performed in [33], by defining several relationships between testing criteria, namely *narrows*, *covers*, *partitions*, *properly covers*, and *properly partitions*. The probability of causing at least one failure is used to measure the different performances of the methods. Moreover, other analytical comparisons have been conducted through Majorization and Schur functions [12], and by other measurements, such as, the expected number of failures caused [78].

As all these simulations and formal analysis have been performed for the purpose of comparison of testing effectiveness and completeness of different testing strategies, testing data from real-world applications are needed to complement and validate the current findings or results.

2.3 Summary

In this chapter, we survey the background, current techniques or strategies, comparisons and evaluations of software fault tolerance and software testing.

Fault-tolerant software enables a system to tolerate software faults remaining in the system after its development. When a fault occurs, fault-tolerant software techniques provide mechanisms within the software system to prevent system failure from occurring.

Fault-tolerant software techniques include single version software techniques and multiple version software techniques. There are two main techniques for single version software fault tolerance: checkpointing and exception handling. Three fundamental techniques are available for multi-version fault-tolerant software: recovery block, N-version programming and N self-checking programming. These approaches are also called design diversity.

Various modeling schemes have been proposed to evaluate the effectiveness of fault-tolerant software. Furthermore, different applications and middleware components have been developed to satisfy performance and reliability demands in various domains employing fault-tolerant software. Fault-tolerant software is generally accepted as a key technique in achieving highly reliable software.

On the software testing side, the two major testing strategies are subdomain-based testing and random testing. Functional testing, structural testing and mutation testing are all design by different testing principles in subdomain-based testing. Code coverage has been proposed to be an indicator of testing effectiveness, but this remains controversial and further empirical validation is needed.

From this survey, we can see that although formal analysis, simulations and experiments have been performed for evaluation and comparison of existing software fault tolerance techniques and software testing strategies, there is a lacking of real world project data for investigation on software testing and fault tolerance techniques together, with comprehensive analysis and evaluation.

□ **End of chapter.**

Chapter 3

Research Procedure and Methodology

Software reliability engineering techniques can be classified in the following areas: fault avoidance, fault removal, fault tolerance, and fault prediction. Traditionally, software reliability is achieved by fault avoidance techniques (including structure programming, software reuse, and formal methods) to prevent software faults or by fault removal techniques (including testing, verification, and validation) to detect and eliminate software faults. As the complexity of software increases, the number of dormant software faults present at system operation also increases. Therefore, the capability to tolerate software faults, particularly for critical applications, is evident.

While fault-tolerant software is seen as a necessity, it also remains a controversial technique and there is a lack of conclusive assessment about its effectiveness. 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 become. 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 efforts, it is doubtful that this impasse can be broken.

Consequently, based on our background study and literature review, we propose a comprehensive procedure in assessing fault-tolerant software for software reliability engineering. We will study the testing and modeling techniques for fault-tolerant software, and construct a systematic approach to predicting the achievable reliability based on the software architecture and testing evidences of fault-tolerant software systems.

Although many research efforts have been conducted for investigation, experimentation, modeling and evaluation of software fault tolerance, 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 software fault tolerance; and the other is the failures in software versions may not occur independently, making it difficult to establish justifiable predictive reliability models. This thesis aims at expanding our scientific understanding of software fault tolerance techniques with quantitative assessment for planning, evaluation, and trade-off study purposes. Specifically, we need to scrutinize and rationalize the modeling and measurement aspects of software fault tolerance as an effective software reliability engineering technique.

We will perform four major tasks in *modeling*, *experimentation*, *evaluation*, and *economics* of software fault tolerance, as illustrated in Figure 3.1. We describe them in detail in the following sections.

3.1 Modeling

In the *modeling* area our objective is to derive a comprehensive reliability model for fault-tolerant software. We would like to compare various fault-tolerant software reliability models, establish a new paradigm which can consider the key attributes of fault-tolerant software systems, and formulate the relationship between fault tolerance techniques and reliability achievement.

In this task we will investigate modeling techniques for fault-tolerant software. We would like to compare various fault-tolerant software reliability models, establish a new paradigm which can consider the key attributes of diverse software systems, and formulate the relationship between software testing and reliability achievement.

In the previous chapter, we have surveyed five historical reliability models for fault-tolerant software. To evaluate their applicability to real-world projects and their prediction accuracy, we plan to perform a comprehensive experimentation to compare these reliability models, as stated in the next section. After comparing these five historical models, we plan to propose our own reliability modeling with the consideration of fault-tolerant architectures, fault correlations, input domains, and system dependencies. The resulting model may not be analytically solvable, in which case simulation techniques [36] will be engaged.

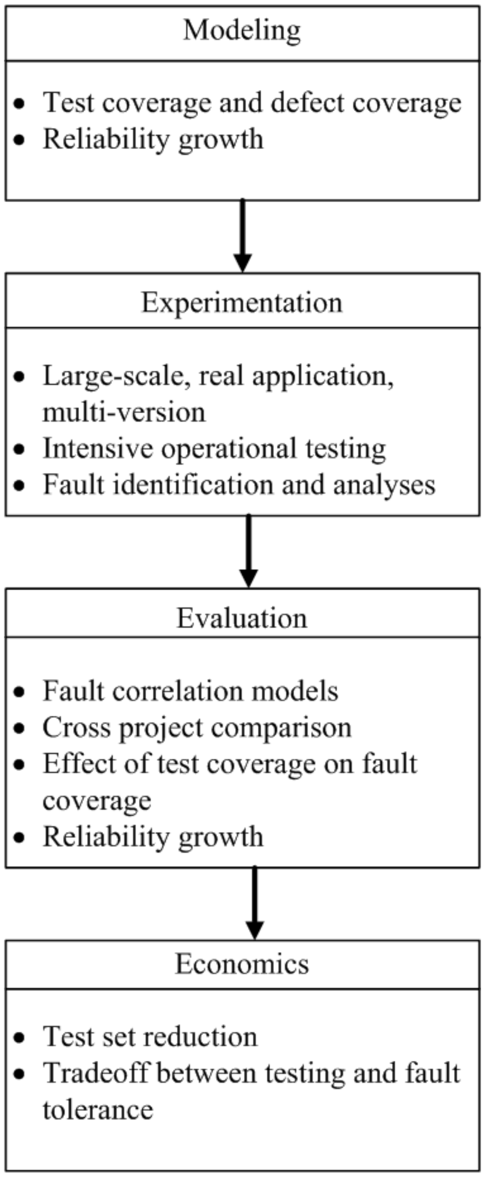
Modeling
• Test coverage and defect coverage
• Reliability growth
Experimentation
• Large-scale, real application, multi-version
• Intensive operational testing
• Fault identification and analyses
Evaluation
• Fault correlation models
• Cross project comparison
• Effect of test coverage on fault coverage
• Reliability growth
Economics
• Test set reduction
• Tradeoff between testing and fault tolerance

Figure 3.1: Four major tasks

3.2 Experimentation

In the *experimentation* area our objective is to obtain new real-world data regarding fault-tolerant software. We plan to conduct fault-tolerant software experiments, apply coverage-based and mutation-based testing techniques, and collect data for detailed analysis. We will conduct comprehensive experimentation to study the nature, source, type, detectability, and effect of faults uncovered in the program versions, and to learn the correlations among these faults and the relation to their resulting failures.

We have pointed out that there is a lack of experimental data for fault-tolerant software. Nevertheless, we have surveyed most fault-tolerant software development experiments, including NASA four-university experiment [27], UCLA six-language experiment [5], University of Iowa Lyu-He Experiment [71]. To further explore the fault-failure relationship, test data effectiveness, fault correlation, etc. for fault-tolerant software, we will conduct a real-world project as our own experiment with considerable size, complexity and of certain representativeness. In addition, we would like to conduct a cross-comparison between our experiment and similar existing ones to investigate some "variants" as well as "invariants" in fault-tolerant software.

3.3 Evaluation

In the *evaluation* area our objective is to provide scientific evidences for software fault tolerance. We intend to apply statistical techniques to confirm the validity of the fault-tolerant software reliability model we established, subject to the evidences from the experimental data. In addition, we will investigate the effectiveness of data flow coverage, mutation coverage, and design diversity for fault coverage. We will examine different hypotheses on software testing and fault tolerance schemes, and establish quantitative evidences regarding the new testing schemes and reliability models that we establish.

3.4 Economics

Traditionally fault-tolerant software and software testing represent two schools of researchers with drastically different opinions on how to obtain reliable software: The former believe software will never be produced free of faults, thus fault tolerance is inevitable, while the latter consider software testing is more cost-effective in building reliable software [104], even for critical applications. Few research efforts were devoted to establish a direct comparison between these two approaches with scientific arguments and quantitative investigations.

In the *economics* area our objective is to perform a trade-off study between the two major software reliability engineering techniques: software fault tolerance and software testing. We can attempt to establish the relationship between software fault tolerance and software testing, and determine cost-effectiveness in these two competing mechanisms. The established relationship between fault tolerance and software testing can be parameterized by the experimental data we obtained for a quantitative assessment for software reliability engineering economics. Other investigations also include impact the existence of fault-tolerant software to software testing [67], and the testing techniques for fault-tolerant software systems [69]. Our ultimate goal is to establish a quantitative relationship between testing techniques and software fault tolerance techniques for trade-off purposes.

3.5 Summary

In this chapter, we describe the research procedure and methodology which outlines the whole work in this thesis. As discussed above, we would like to perform four major procedures regarding software reliability engineering, software fault tolerance and software testing, namely, *modeling*, *experimentation*, *evaluation*, and *economics*.

The significance of this research is its potential penetration to a long-term research problem in assessing fault-tolerant software as a validated technique for software reliability engineering purpose. It will engage a number of other reliability engineering efforts, including reliability modeling, software testing strategies, test case development, and feedback control for optimal software architecture design. The new models, new data, and new evidences obtained from this research work can promote a systematic approach of fault tolerance software in modern systems.

In the following chapters, we will present our work on experimentation (Chapter 4), evaluation (Chapter 6&7) and modeling (Chapter 5&8). The economics part will be put as our future work (Chapter 9).

□ **End of chapter.**

Chapter 4

Experimental Setup and Data Collection

To investigate and evaluate the reliability and fault correlation features of N-version programming, statistical failure data are highly demanded from experiments or real-world projects. To simulate real environments, the experimental application should be as complicated as real-world projects to represent actual software in practice. In such experiments, the population of program versions should be large enough to provide valid statistical analysis. Furthermore, the development process should be well-controlled, so that the bug history can be recorded and real faults can be studied.

Up to now, a number of projects have been conducted to investigate and evaluate the effectiveness of N-version programming, including UCLA Six-Language project [54, 69], NASA 4-University project [27, 86, 103], Knight and Leveson's experiment [58], and Lyu-He study [25, 71]. Considering the population of programming versions and the complexity of the application, NASA 4-University project was a representative and well-controlled experiment for the evaluation of N-version programming.

Our research is motivated by the lack of real world project data for investigation on software testing and fault tolerance techniques together, with comprehensive analysis and evaluation. Subsequently we conducted a real-world project and engaged multiple programming teams to independently develop program versions based on an industry-scale avionics application. We conducted detailed experimentation to study the nature, source, type, detectability, and effect of faults uncovered in the program versions, and to learn the relationship among these faults and the correlation of their resulting failures. We applied the mutation testing techniques to reproduce mutants with *real* faults, and investigated the effectiveness of data flow coverage, mutation coverage, and design diversity for fault coverage.

In this chapter, we demonstrate the experimental setup and preliminary data collected. Some of the data and results will be further analyzed in the following Chapters.

4.1 Project Descriptions and Experimental Procedure

As stated before, our research is motivated by the lack of real world project data for the investigation and evaluation of the effect of code coverage on fault detection capability with current software testing strategies. For the purpose of evaluation and comparison, mutation testing provides a testing adequacy criterion in unit testing [82, 83, 109], integration testing [23] and program analysis [84]. Moreover, this testing adequacy criterion will be more realistic if real faults rather than hypothetical faults are seeded into the original program versions.

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

4.1.1 RSDIMU Project

The specifications for a critical avionics instrument, Redundant Strapped-Down Inertial Measurement Unit (*RSDIMU*), were used in our project investigation. RSDIMU was first engaged in [27] 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 is fault tolerant, it allows the calculation of the acceleration when some of the accelerometers fail. The specification allows for the accelerometers to fail before the beginning of the program or during the execution of the program. Figure 4.1 shows the system data flow diagram.

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

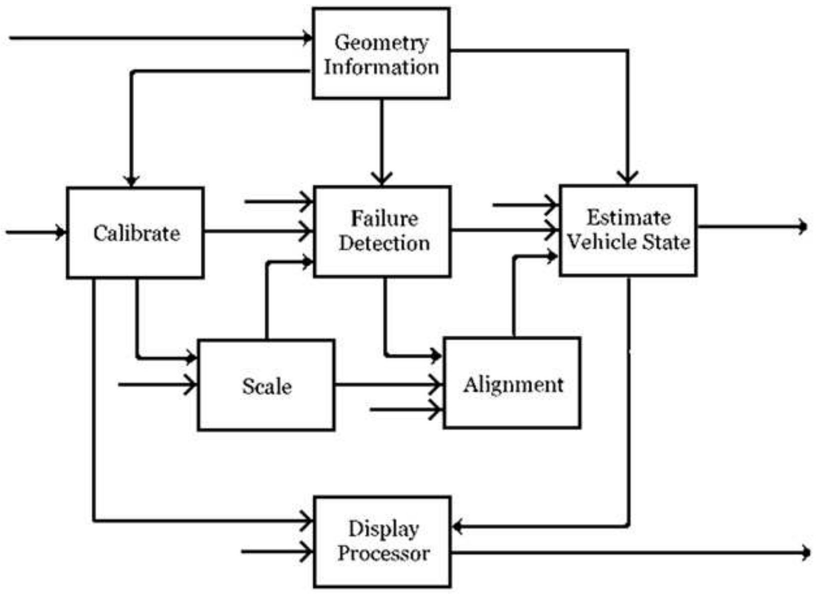

Figure 4.1: RSDIMU system data flow diagram

("Calibrate") and scaling ("Scale").

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

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

4.1.2 Software Development Procedure

The waterfall model [93] was applied in this software development project. Six phases were conducted in the development process according to the software engineering requirements.

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

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

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

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

Phase 3: Initial code (duration: 1.5 weeks)

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

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

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

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

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

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

Programmers formally submitted their programs for a stringent acceptance test, where 1200 test cases were used to validate the final code. At the end of this phase all 34 teams passed the acceptance test. It is noted that the requirement for this acceptance test was the same as the operational test conducted in [27], which was much tougher than the original acceptance test in [27].

4.1.3 Mutant Creation

A Revision Control System (RCS) [92] was required for source control for each team. Every code change of each program file at each check-in could therefore be identified. Software faults found during each stage were also identified. These faults were then injected into the final program versions to create mutants, each contain one programming fault. We selected 21 program versions for detailed investigation, and created 426 mutants. We disqualified the other 13 versions as their developers did not follow the development and coding standards which were necessary for generating meaningful mutants from their projects.

The following rules were applied in the mutant creation process:

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

2. Some changes only occurred in middle versions. For example, the changes between v1.1 and v1.2 may not be completely manifested in the final version. These changes were then ignored.

3. Temporary code changes for debugging purposes were not included.

4. Modifications of the function prototypes were excluded.

5. As the specification does not mention memory leaks, mutants were not created to generate any faults leading to memory leaks.

6. The same programming error may span many blocks of code. For example: a variable should be divided by 1000 when it is used. The missing division may occur everywhere in the source files. This would be counted as a single fault.

4.1.4 Setup of Evaluation Test

In order to evaluate the effectiveness of data flow testing schemes, we set up an evaluation test environment. We employed the ATAC (Automatic Test Analysis for C) [45, 72] tool to analyze and compare code coverage achieved in testing conducted with the 21 program versions, as well as their 426 mutants. For each round of evaluation test, all 1200 acceptance test cases were exercised on these mutants. This was a very intensive testing procedure, as all the resulting failures from each mutant were analyzed, their coverage measured, and cross-mutant failure results compared.

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

The test cases conducted in the evaluation test are described in Table 4.1. Based on execution of these test cases over the mutants, we analyzed the relationship between fault and failure. We examined the effectiveness of the test cases by their test coverage measures, and by their ability to kill the

Table 4.1: 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.

mutants (a mutant is killed if it gives incorrect outputs compared with the outputs from a gold version). We also studied the fault detecting capability of each test case, and obtained the non-redundant set of test cases covering all mutants.

4.1.5 Program Metrics

Table 4.2 shows the program metrics for the 21 versions engaged in the evaluation test, and the mutants each of them generated. It can be noted that the size of these programs varies from 1455 to 4512 lines of source code. Each version produced a number of mutants ranging from 9 to 31. The data flow metrics are also listed in Table 4.2.

Table 4.2: Program metrics for 21 versions

Id	Lines	Modules	Functions	Blocks	Decisions	C-Use	P-Use	Mutants
01	1628	9	0	1327	606	1012	1384	25
02	2361	11	37	1592	809	2022	1714	21
03	2331	8	51	1081	548	899	1070	17
04	1749	7	39	1183	647	646	1339	24
05	2623	7	40	2460	960	2434	1853	26
07	2918	11	35	2686	917	2815	1792	19
08	2154	9	57	1429	585	1470	1293	17
09	2161	9	56	1663	666	2022	1979	20
12	2559	8	46	1308	551	1204	1201	31
15	1849	8	47	1736	732	1645	1448	29
17	1768	9	58	1310	655	1014	1328	17
18	2177	6	69	1635	686	1138	1251	10
20	1807	9	60	1531	782	1512	1735	18
22	3253	7	68	2403	1076	2907	2335	23
24	2131	8	90	1890	706	1586	1805	9
26	4512	20	45	2144	1238	2404	4461	22
27	1455	9	21	1327	622	1114	1364	15
29	1627	8	43	1710	506	1539	833	24
31	1914	12	24	1601	827	1075	1617	23
32	1919	8	41	1807	974	1649	2132	20
33	2022	7	27	1880	1009	2574	2887	16
Average	2234.2	9.0	48.8	1700.1	766.8	1651.5	1753.4	Total: 426

Table 4.3: Defect type distribution

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

4.2 Static Analysis of Mutants: Fault Classification and Distribution

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

4.2.1 Mutant Defect Type Distribution

Each mutant is assigned with a defect type according to [20]. The statistics is show in Table 4.3.

4.2.2 Mutant Qualifier Distribution

Each mutant is assigned with a qualifier. The statistics is show in Table 4.4, with the following definitions:

Table 4.4: Qualifier distribution

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

Table 4.5: Severity distribution

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

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

4.2.3 Mutant Severity Distribution

The severity distribution according to the following definitions is listed in Table 4.5.

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

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

Table 4.6: Development stage distribution

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

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

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

Note that in Table 4.5, "Highest Severity" records the highest level of severity among all failed test cases for a mutant, while "First Failure Severity" records the failure severity at the first time when a failure occurred to the mutant.

4.2.4 Fault Distribution over Development Stage

The sources of faults came from different stages of the development. This distribution is shown in Table 4.6.

4.2.5 Mutant Effect Code Lines

The number of code lines span affected by each mutant was measured by manual inspection. Table 4.7 lists the details. In previous research efforts on mutation testing, usually the faults were artificially injected which simple code

Table 4.7: Fault effect code lines

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

changes such as the replacement of a logic operator in a conditional statement or the modification of a operand value, and the code line span was limited to one or a few lines. It can be seen from Table 4.7 that, in our experiment, an average of 11.39 code lines were affected by a fault, accurately resembling genuine faults and showing that artificial mutants are not representative of real situations.

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

Based on execution of the 1200 test cases over the mutants, we analyzed fault and failure relationship. We also studied the fault detecting capability of each test case, and obtained the non-redundant set of test cases which can cover all mutants.

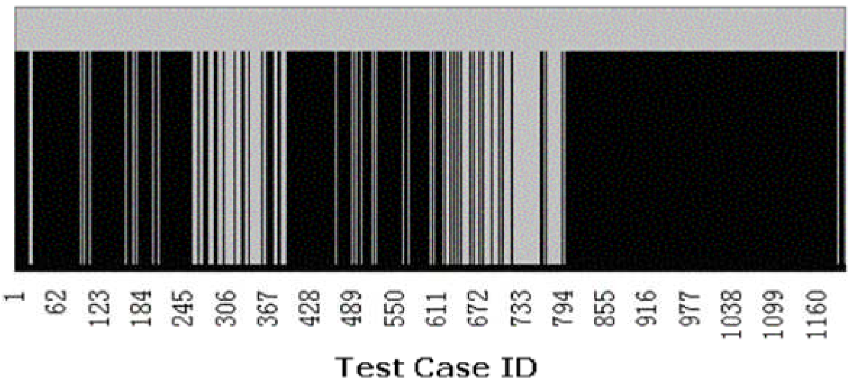

Figure 4.2: Non-redundant set of test cases

4.3.1 Finding Non-redundant Set of Test Cases

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

Figure 4.2 shows the non-redundant test set from the 1200 test cases. The gray lines indicate redundant cases, while the black blocks indicate the set of non-redundant test cases. The size of this test set is 698 test cases.

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

4.3.2 Relationship between Mutants

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

Table 4.8: Mutants relationship

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

three mutant relations can be defined:

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

Table 4.8 shows distribution of these mutant relations, and their percentages out of total combinations (90525).

4.3.3 Relationship between the Programs with Mutants

During the evaluation test, we also determined the correlation among the program version based on mutant executions. We defined two types of relationships: program versions with similar mutants, and program versions with exact mutants. The former includes program versions which generate similar mutants, while the latter includes those generating exact mutants. The results are shown, respectively, in Table 4.9 and Table 4.10. Each axis in these tables shows the program ID, and the values in the content, if any, indicate the number of similar or exact mutants between two corresponding program versions. Note these tables are symmetric.

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

Here are the descriptions on the causes of these faults:

Pair 1 – Versions 4 and 8

The display mode is incorrectly calculated for a missing operation.

Pair 2 – Versions 12 and 31

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

Pair 3 – Versions 15 and 33

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

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

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

Pair 7 – Versions 31 and 32

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

We note that the amount of exact faults among program versions is very limited. This implies that design diversity involving multiple program versions

Table 4.9: Program versions with similar mutants

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

Table 4.10: Program versions with exact mutants

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

Table 4.11: Summary of program relationship

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

Table 4.12: Exact pair 1: versions 4 and 8

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

Table 4.13: Exact pair 2: versions 12 and 31

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

Table 4.14: Exact pair 3: versions 15 and 33

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

Table 4.15: Exact fault pairs 4, 5, and 6: versions 4, 15 and 17

	Version 4	Version 15	Version 17
Module	Estimate Vehicle State	Estimate Vehicle State	Estimate Vehicle State
Stage	Initcode	Initcode	Initcode
Defect Type	Assign/Init	Assign/Init	Algorithm/Method
Severity	B	B	B
Qualifier	Incorrect	Incorrect	Incorrect

Table 4.16: Exact pair 7: versions 31 and 32

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

can be an effective mechanism for software reliability engineering.

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

4.4 Threats to Validity

Internal as well as external threats to validity may arise in empirical studies. The former are related to the consistency of the measurement, the appropriate use of tools and methods, etc. The latter touch on the issue of the extent to which the present results can be applied to other studies.

First of all, a main threat may arise from the representativeness of this particular application, compared with real projects in business. Although the RSDIMU application is computational-intensive and contains no graphic interface or user interactions, unlike most software industry projects nowadays, it is an important part of the navigation system of spacecraft; it is mission-critical and computational intensive, which requires extreme high reliability for the software. We believe it can represent general business projects based on its complexity, size and well-controlled development process.

Another threat may arise from the development of the multiple versions of the same program: are they really developed independently? In our project, the policy prohibits the students from joint work and sharing between teams. The students can ask questions on the course newsgroup, but only about the specification. Of course, because of the similar background and programming experience among the different team members, some common faults exist in the development and testing phases. However, as identified and analyzed in this chapter, such common faults are few in number.

A possible question may be related to the developers' incentives and work experience. Although all the programs were developed as a course project, the developers tried their best to pass all the different testing phases as the project is heavily weighed in the course work. Also we think the work experience of the developers do not account that much in this application, as it contains comprehensive computations which are clearly formulated in the specification.

Furthermore, in this study, we use the automated tool ATAC to collect the code coverage information in testing both the individual coverage for each test case, and the cumulated coverage for the whole test set. As one of the most popular coverage collecting tools for the C/C++ programming language, ATAC has been adopted in various studies for coverage testing and collecting

[15, 76, 107]. We use ATAC to obtain the four measures of code coverage: block, decision, C-use and P-use.

For the external threats to validity, we believe the results can be applied to other fault-tolerant related studies, especially for mission-critical software systems with similar features. Moreover, the evaluations on testing strategies and reliability modeling can also be applied to general software systems, in case that detailed fault and code coverage data can be collected.

4.5 Summary

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

Our results implied that design diversity involving multiple program versions can be an effective solution for software reliability engineering, since the portion of program versions with exact faults is very small. The quantitative tradeoff between these two approaches, however, remains a research issue. Currently we can only generally perceive that software fault removal and software fault tolerance are complementary rather than competitive.

All the data collected in this experiment will be used for further investigation for various features of design diversity and coverage-based testing strategies in the following chapters.

□ **End of chapter.**

Chapter 5

Evaluations on Reliability Models under Fault Correlation

In design diversity, with multiple "independently developed" program versions, one would expect that failures in a subset of the versions may be masked or at least detected; coincident failures of all versions will be less frequent than failures of any single version; and thus a multiple-version system will fail less often than a single version. One might hope that the different versions fail "independently", but in some empirical studies failures of multiple versions were positively correlated [9, 58].

As coincident failures may exist in diverse systems, and faults between multiple software versions may correlated with each other, the reliability of the final diverse systems may not be so good as expected. Some methods have been proposed to attempt the modeling of reliability and fault correlations achieved in design diversity, as have surveyed in Chapter 2. The five historical reliability models try to formulate the reliability of diverse software systems with respect to single software system, using various mathematical methods such as probability modeling and fault tree.

In this chapter, we evaluate existing reliability models under fault correlation with design diversity using the data collected in our experiment, as described in Chapter 4. We will apply two main fault correlation models, i.e., Popov, Strigini et al model and Dugan and Lyu Model, on our generated mutants and evaluate their effectiveness.

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

Popov, Strigini et al's model (PS model) [86] 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

$$P_{A,B|S_i} = P_{A|S_i} P_{B|S_i} + cov_i(\Omega_A, \Omega_B). \tag{5.1}$$

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

$$P_{(A,B)} \leq \sum_i \min{(P_{A|S_i}, P_{B|S_i})} P(S_i). \tag{5.2}$$

Table 5.1: Alternative expressions for the pfd of a 1-out-of-2 system (from [86])

$\sum_{x \in D} \omega_A(x) \cdot \omega_B(x) \cdot P(x)$				
$P_A \cdot P_B$ (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)			$+$	$E(cov_i(\Omega_A, \Omega_B))$
$P_{A,B_{sub-ind}}$			$+$	$E(cov_i(\Omega_A, \Omega_B))$

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

$$P_{A,B_{sub-ind}} = \sum_i P_{A|S_i} P_{B|S_i} P(S_i), \tag{5.3}$$

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

Alternative expressions for $P_{A,B}$ as the *pfd* of a 1-out-of-2 version system are given in Figure 5.1.

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.

5.1.1 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 [27] 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 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 [27] 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

[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.

Table 5.2: 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

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 5.2. Note that the six mutants are from different initial versions with injection of different design and programming faults.

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 5.3. The former three in Table 5.3 are hypothetical demand files described in [86], 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 [86], Popov, Strigini et al discuss the use of both observed frequencies and of conservative confidence bounds as estimates of the conditional *pfds*, and favor the second alternative. Particularly in our case, according to Table

Table 5.3: 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 5.4: 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

5.2, no failure was observed in some subdomains. Thus we adopt confidence bounds method to estimate the joint *pfds* in our experiment. Table 5.4 shows the 90% confidence upper bounds on *pfds* of mutants in subdomains, and Table 5.6 displays the lower bounds. Our testing results for upper bounds and lower bounds on joint *pfds* under four demand profiles are listed in Table 5.5 and Table 5.7, respectively.

Table 5.5: 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
(215, 382)		$P_{215\_90\%}$	$P_{382\_90\%}$	$\min(P_{215\_90\%}, P_{382\_90\%})$	$P_{215,382_{upper90\%}}$
	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
(382, 403)		$P_{382\_90\%}$	$P_{403\_90\%}$	$\min(P_{382\_90\%}, P_{403\_90\%})$	$P_{382,403_{upper90\%}}$
	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 5.6: 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

Table 5.7: 90 percent confidence lower bounds on the joint pfds under demand profiles

Pair		P_{117}	P_{305}	$cov(S_{117}, S_{305})$	$P_{117}P_{305}$	$P_{117,305_{sub\_ind}}$
(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	$5.87 \cdot 10^{-3}$	$7.79 \cdot 10^{-3}$	$-2.47 \cdot 10^{-5}$	$4.57 \cdot 10^{-5}$	$2.09 \cdot 10^{-5}$
(215, 382)		P_{215}	P_{382}	$cov(S_{215}, S_{382})$	$P_{215}P_{382}$	$P_{215,382_{sub\_ind}}$
	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	$2.83 \cdot 10^{-3}$	$1.63 \cdot 10^{-2}$	$1.70 \cdot 10^{-4}$	$4.62 \cdot 10^{-5}$	$2.16 \cdot 10^{-4}$
(382, 403)		P_{382}	P_{403}	$cov(S_{382}, S_{403})$	$P_{215}P_{382}$	$P_{382,403_{sub\_ind}}$
	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	$1.63 \cdot 10^{-2}$	$5.86 \cdot 10^{-3}$	$-1.16 \cdot 10^{-5}$	$9.56 \cdot 10^{-5}$	$8.40 \cdot 10^{-5}$

5.1.2 Comparison and Discussion

The target objects engaged in our experiment and NASA 4-university experiment studied in [86] 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 (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 5.7 is a small positive number under DP1, while negative in the other three demand profiles. Thus the "likely" lower bound $P_{117,305_{sub\_ind}}$ 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 $P_{215,382upper}$ 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) 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 [86], 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 [86] 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 [86] 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 [86] 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.

5.2 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 [25]. 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 5.1. The three parameters are calculated by the following equations for 3-version systems:

$$P_V = \frac{F_1}{NF_0 + F_1}, \tag{5.4}$$

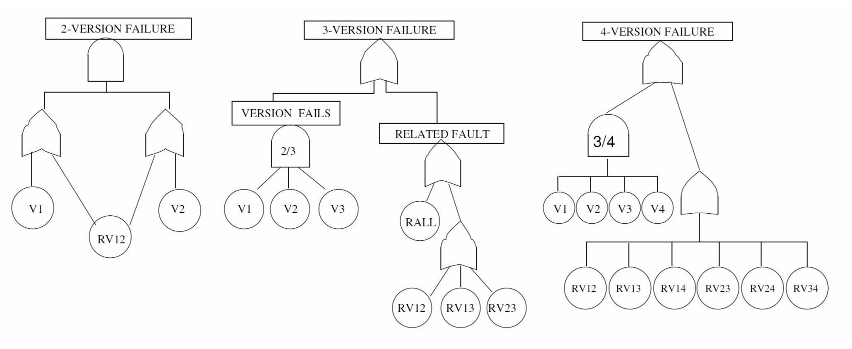

Figure 5.1: Fault tree models for 2, 3 and 4 version systems (from [25])

$$P_{RV} = \frac{2F_2P_V(1-P_V) - (N-1)F_1P_V^2}{2F_2P_V(1-P_V) + (N-1)F_1(1-P_V^2)}, \tag{5.5}$$

$$P_{RALL} = \frac{F_3 - {P_V}^3}{1 - {P_V}^3}, \tag{5.6}$$

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 [25]. 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 in Table 5.8. We can see from Table 5.8 that the average failure probability for single version is 0.01, which is much smaller compared with the original experimental data in [25]. 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.

We configure the six mutants in pairs, and compare their outputs for each test case. Table 5.9 yields an estimate of $P_V = 0.0084$ for the probability

Table 5.8: 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

Table 5.9: 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

of raising an unrelated failure in a 2-version configuration, and an estimate $P_{RV} = 0.0016$ for the probability of a related failure.

Next, the six mutants are configured in sets of three. Table 5.10 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 5.11. Unlike the previous experiment reported in [24], 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.

The mutants are then analyzed in combinations of four programs. Table

Table 5.10: 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 5.11: 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

5.12 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 5.13. 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 5.14 summarizes the parameters estimated from our data. The parameters are applied to the fault tree model shown in Figure 5.1. 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

Table 5.12: 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 5.13: 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

Table 5.14: 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

close to zero but the observed frequency is 0.0015. Our 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.

Figure 5.2 compares the predicted reliability of three different configurations, including 2-version configuration for Distributed Recovery Block (DRB) [89], 3-version configuration for N-Version Programming (NVP) [2, 71], and 4-version configuration for N-Self Checking Programming (NSCP) [59]. 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 [24], the prediction performance of the three configurations in our experiment are consistent with those in [24]. However, if we look into the first hundreds of hours, the three configurations performs differently, as shown Figure 5.3. Here NSCP depicts higher reliability than DRB and NVP, although it gives the least reliability in the long run.

Figure 5.4 compares the predicted safety of the three systems. Here we assume that the decider used in the NVP and NSCP has a failure probability

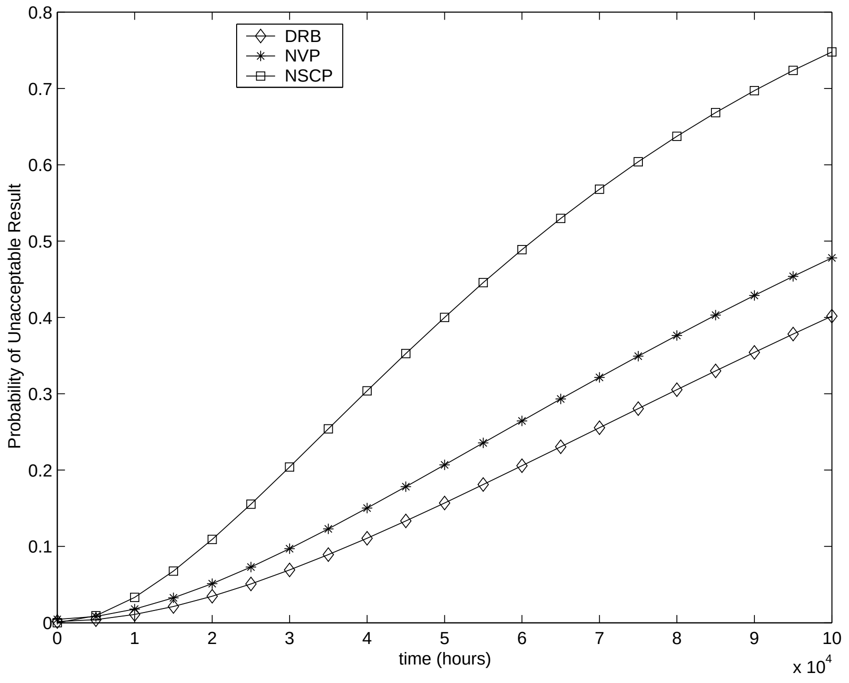

Figure 5.2: Predicted reliability by different configurations

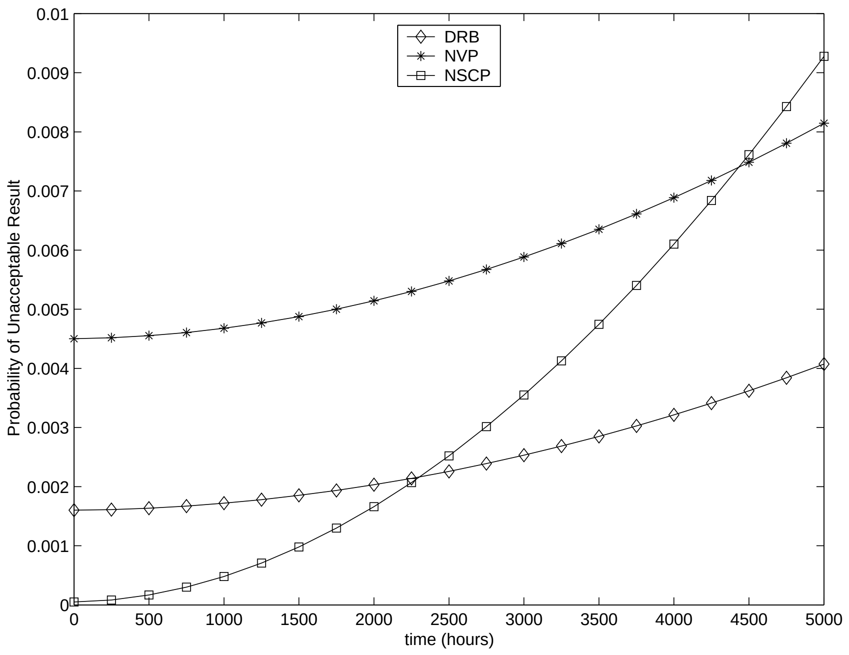

Figure 5.3: Predicted reliability by different configurations

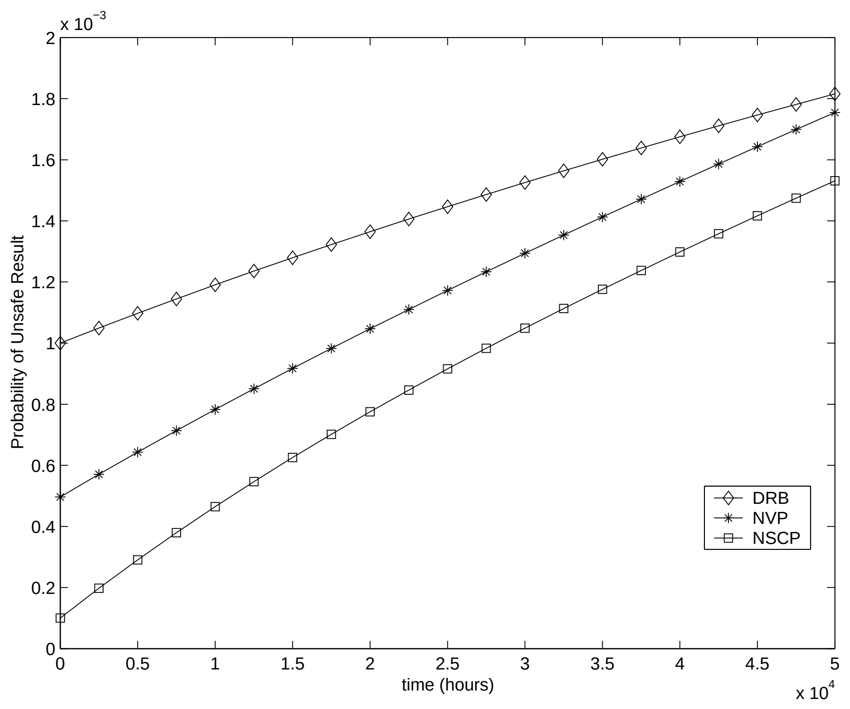

Figure 5.4: Predicted safety by different configurations

of only 0.0001 and that for DRB has a failure rate of 0.001 [25]. According to Figure 5.4, 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 [25].

Overall, compared our project with former project in [25], 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 [25] and the fault tree modeling. To achieve more accurate results, the information about the correlation between successive executions should be included [95].

5.3 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 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 reliability 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.

□ **End of chapter.**

Chapter 6

Cross Project Comparison on Reliability Features

The main issue with diverse systems is how to predict the final reliability as well as how to estimate the fault correlation between multiple versions. To investigate this, theoretical as well as empirical investigations have been conducted based on experimentation [27, 38, 39, 71, 74], modeling [25, 28, 29, 64, 97], and evaluation [8, 65, 66, 86] for reliability and fault-correlation features of N-version programming.

In this chapter, we perform comprehensive operational testing on our original versions and collect statistical data on coincident failures and faults. We also conduct comparison with NASA 4-University project [27] and search for the "variant" as well as "invariant" features in N-version programming between our project and NASA 4-University project, two projects with the same application but separated by 17 years. Both qualitative and quantitative comparisons are engaged in development process, fault analysis, average failure rate and reliability improvement of N-version programming.

The terminology used for experimental descriptions and comparisons is defined as the following. *Coincident failures* refer to the failures where two or more different software versions respond incorrectly (with respect to the specification) on the same test case, no matter whether they produce similar or different results. *Related faults* are defined as the faults which affect two or more software versions, causing them to produce coincident failures on the same test cases, although these related faults may not be identical.

6.1 Experimental Background

The experimental procedures of both NASA 4-University project and our project are described in the following subsections.

6.1.1 NASA 4-University Project

In 1985, NASA Langley Research Center sponsored an experiment to develop a set of high reliability aerospace application programs to study multi-version software approaches, involving graduate student programmers from four universities in 20 program teams [103]. Details of the experimental procedure and development process can be found in [27, 54, 103].

In this experiment, the development process was controlled to maintain as much independence as possible between different programming teams. In addition, the final twenty programs went through a separate three-phase testing process, namely, a set of 75 test cases for acceptance test, 1196 designed and random test cases for certification test, and over 900,000 test cases for operational test.

The testing data collected in NASA 4-University project have been widely investigated to explore the reliability features of N-version programming [27, 64, 86].

6.1.2 Our Project Descriptions

In the Spring of 2002 we formed 34 independent programming teams at the Chinese University of Hong Kong to design, code, test, evaluate, and document the RSDIMU application. Each team was composed of four senior-level undergraduate Computer Science students for a twelve-week project in a software engineering course. The project details and the software development procedure are portrayed in Chapter 4.

The acceptance test set contains 1196 test cases, including 800 functional test cases and 400 randomly-generated test cases. Another random 100,000 test cases have been conducted recently in an operational test for a final evaluation of all the 34 program versions in our project.

6.2 Qualitative Comparison with NASA 4-University Project

In NASA 4-University project [54, 103], the final 20 versions were written in Pascal, while developed and tested in a UNIX environment on VAX hardware. As this project has some similarities and differences with our experiment, interesting observations can be made by comparing these two projects, which are widely separated both temporally and geographically, to identify possible "variants" as well as "invariants" of design diversity.

The commonalities and differences of the two experiments are shown in Table 6.1. The two experiments engaged the same RSDIMU specification, with the difference that NASA 4-University project employed the initial version of the specification which inherited specification incorrectness and ambiguity, while we employed the latest version of the specification and little specification

Table 6.1: Comparisons between the two projects

Features	NASA 4-University project	Our experiment
Commonality		
1. same specification	initial version (faults involved)	mature version
2. similar development duration	10 weeks	12 weeks
3. similar development process	training, design, coding, testing, preliminary acceptance test	initial design, final design, initial code, unit test, integration test, acceptance test
4. same testing process	acceptance test, certification test, operational test	unit test, integration test, acceptance test, operational test
5. same operational test environment (i.e., determined by the same generator)	1196 test cases for certification test	1200 test cases for acceptance test
Difference		
1. Time (17 year apart)	1985	2002
2. Programming Team	2-person	4-person
3. Programmer experience	graduate students	undergraduate students
4. Programmer background	U.S.	Hong Kong
5.Language	Pascal	C

faults were detected during the development. The development duration were similar: 10 weeks vs. 12 weeks. The development process and the testing process are also similar. We engaged 1200 test cases before accepting the program versions, which were then subjected to 100,000 cases for operational test. NASA 4-University project involved the same 1196 test cases in its certification test, and other 900,000 test cases as operational test. Note that all test cases were generated by the same test case generator. The two experiments, on the other hand, differed in time (which is 17 years apart), programming team (2-person vs. 4-person), programmer experience and background, as well as the programming language employed (Pascal vs. C).

6.2.1 Fault Analysis in Development Phase

As the two N-version programming experiments exhibit native commonalities and differences, it will be interesting to see what remains unchanged in these two experiments concerning software reliability and fault correlation. As stated above, the original version of the RSDIMU specification involved some faults which were fixed during NASA 4-University project development. We first consider the design and implementation faults detected during the NASA certification test and our acceptance test. These two testing phases employed the same set of test cases, and a comparison between them should be reasonable. In our investigation, we focus our discussion on the related faults, i.e., the faults which occurred in more than one version, and triggered off coincident failures of different versions at the same test case.

The classification of the related faults in our experiment is listed in Table 6.2. There are totally 15 categories of related faults. The distribution of the related faults is listed in Table 6.3. All the faults are design and implementation faults. Only class F1 is at low severity level; others are at critical severity

Table 6.2: Related faults detected in our experiment

Faults	Brief Description	severity level
F1	Display error	low
F1.1	improper initialization and assignment	
F1.2	incorrect rounding at Display values	
F1.3	DisplayMode assignment error	
F2	Misalignment problem	critical
F2.1	the misalignment should be in radians	
F2.2	the misalignment should be in milliradians	
F2.3	wrong frame of reference	
F3	Sensor Failure Detection and Isolation problems	critical
F3.1	fatal failures due to initialization problem	
F3.2	wrongly set of the status of system	
F3.3	update problem after failure detection	
F3.4	test threshold for Edge Vector Test is miscalculated or misjudged	
F3.5	wrong OFFRAW array order	
F3.6	arithmetic and calculation error	
F4	Acceleration estimation problem	critical
F4.1	specific force recalculated problem after failure detection	
F4.2	wrongly calculation with G (should multiply 0.3048)	
F5	Input from sensor not properly masked to 12 bits (e.g., “mod 4096” missing)	critical

Table 6.3: Distribution of related faults detected

Faults	Versions	Fault Span
F1		
F1.1	P2,P4,P9	3
F1.2	P4,P8,P15,P17,P18	5
F1.3	P1,P2,P4,P5,P7,P8,P9,P15,P17,P18,P24,P27,P33	13
F2		
F2.1	P3	1
F2.2	P1,P8,P15	3
F2.3	P15	1
F3		
F3.1	P3,P4,P5,P7,P8,P22,P27	7
F3.2	P1,P8	2
F3.3	P1,P2	2
F3.4	P31,P32	2
F3.5	P12,P31	2
F3.6	P1,P5	2
F4		
F4.1	P2,P18	2
F4.2	P4,P12,P15,P17	4
F5	P12,P17,P15,P33	4

level.

Comparing Table 6.2 with the related faults detected in certification testing of the NASA project [103], we can see that some common faults were generated during the development of these two projects. They are F1(D4 in [103]), F2.3(D1.3), F3.1(D3.7), F3.4(D3.1) and F5(D8). F1 faults are related to the display module of the application. Display algorithm was clearly stated in the RSDIMU specification, but it encountered related faults most frequently in both projects. This may be due to the fact that the module was comparatively simple and less critical in the whole application, and programmers inclined to overlook it in both development and testing phases. Fault F2.3 is related to a calculation in the wrong frame of reference, which involved only one version in both experiments. Fault F3.1 is a fault involved in many program versions, causing fatal failures due to initialization problems. Fault F3.4 is similar to fault D3.1, and both were due to misunderstanding of the content of the specification. Finally, the missing process of masking input from sensors to 12 bits (e.g., mod 4096) results in fault F5, which involved four teams in our project and 11 teams in the NASA project.

For the related faults occurring in both projects, some of them (F1, F2.3, F5) were due to misunderstanding of the specification or inadequate efforts spent on the difficult part of the problem, and others (F3.1 and F3.4) were caused by lack of knowledge in the application area or in the programming language area, e.g., omission or wrong variable initialization.

Some fault types occurred in the NASA project but did not occur in our experiment, e.g., D5 (division by zero on all failed input sensors), D6 (incorrect conversion factor) and D7 (Votelinout procedure call placement fault and/or error in using the returned values). Some of the reasons are: 1) As mentioned above, the initial specification contained incorrectness and inconsistency. Some design and implementation faults (e.g., D7) may be caused by the ambiguity or wrong statements in the specification. 2) Certain exception faults, such as division by zero, did not occur in our experiment. This is an interesting phenomenon, and the possible reason is that nowadays students learned the principle of avoiding exception faults as a common practice in the programming language courses.

From the comparison of the two experiments, we can see that both cause and effect of some related faults remain the same. As the application can be decomposed into different subdomains, related faults often occurred in most difficult parts. Furthermore, no matter which programming language was used, the common problems with programming remained the same, e.g., the initialization problem. Finally, the most fault-prone part being the easiest part of the application (i.e., Display module) confirms that a comprehensive testing and certification procedure towards the easiest module is important.

6.2.2 Fault Analysis in Operational Test

As all the 34 versions have already passed the acceptance test, their failures revealed in the operational test deserve to be scrutinized. To investigate the features of these failures, especially those coincident failures occurring in more than two versions, we identify all the operational faults in each version and list them in Table 6.4. There are totally six faults detected during operational test. We denote each individual fault by the version number in which the fault occurs, followed by a sequence number. For example, version 22 contains one single fault 22.1, and version 34 is associated with three different faults: 34.1, 34.2, and 34.3. Table 6.4 also shows the input condition where a corresponding fault is manifested, together with a brief description of the fault.

Since these four versions have already passed the acceptance test, their

Table 6.4: Fault description during operational test

Version	Fault	Input condition	Fault description
22	22.1	At least one sensor fail during the test	Incorrect calculation in sensor failure detection process
29	29.1	No sensor fail before the test, and more than two bad faces due to noise and failure detection, but at least one sensor pass the failure detection algorithm	Omission of setting all sensor failure output to TRUE when the system status is set to FALSE
32	32.1	Three or four sensors fail in more than two faces, due to input, noise, or failure detection algorithm	Incorrectly setting system status to FALSE when more than five sensors fail
34	34.1	Sensor failure in the input	Wrongly setting the second sensor as failure when the first sensor in the same face fails
	34.2	No sensor fail due to input and noise	Incorrect calculation in Edge Vector Test is wrong
	34.3	At most two faces fail due to input and noise, no. of bad faces is greater than 2, and at least one more sensor on the third face fail in failure detection algorithm	Only counting the number of bad face prior to the Edge Vector Test

faults were detected by some extreme situations. It is noticed that these faults are all sensor Failure Detection and Isolation problems, i.e., F3 category in Table 6.2, including wrong setting of system status (F3.2) and arithmetic and calculation error (F3.6). These faults were triggered only under special combinations of individual sensor failures due to input, noise and failure detection process.

In Table 6.4, versions 22, 29 and 32 exhibit single faults, while version 34 contains multiple faults. There is no related fault or coincident failure among the former three versions. For version 34, one of its faults (34.2) is related to the fault in version 22 (22.1), resulting in coincident failures on 25 test cases, as shown in Table 6.6. The other fault 34.3 is related to the one in version 29

(29.1), leading to 32 coincident failures. Although causing coincident failures, these two fault pairs are quite different by nature.

Compared with other program versions, version 34 shows the lowest quality in terms of its program logic and design organization. Particularly, hard code is found in the source code, i.e., some intermediate result was manually assigned according to specific input data of a particular test case, but not through the required computational functions. Because of its lack of detailed report and poor quality, we omitted this version (as well as a number of other versions) when applying mutation testing in our previous study [74]. Since related faults and coincident failures exist in this version, we took a pessimistic approach to include it (and all other program versions) in the following analysis. It is noted that the overall performance of N-version programming derived from our data would be much better, if the failures in version 34 are ignored (as no related faults or coincident failures would then be observed in other versions).

6.3 Quantitative Comparison with NASA 4-University Project

In this comprehensive testing study, we generate 100,000 test cases randomly to simulate the operational environment for RSDIMU application. The failures occurring in all these 34 versions are recorded according to their input and output domains. Here we adopt the similar partition method described in [27], i.e., all the normal operations are classified as one of the following six system states exclusively:

$$\begin{aligned} S_{i,j} &= \{i\ sensors\ previously\ failed\ and \\ &\quad j\ of\ the\ remaining\ sensors\ fail \end{aligned}$$

$$| \ i = 0, 1, 2; \ j = 0, 1 \ \}.$$

In addition, we introduce a new category S_{others} to denote all the exceptional operations which do not belong to any of the above six system states. In the following analysis, we partition the whole input and output domain into seven categories for an investigation in software reliability measurement.

6.3.1 Failure Probability and Fault Density

Table 6.5: Failures collected in our project

Version ID	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}	Total	Probability
22	0	210	0	246	0	133	29	618	0.00618
29	0	0	0	0	0	0	2760	2760	0.0276
32	0	0	0	0	0	0	2	2	0.00002
34	0	617	0	407	0	74	253	1351	0.0135
Total no. of failures	0	827	0	653	0	207	3044	4731	0.0473
Conditional average failure probability	0	0.0024	0	0.0017	0	0.0011	0.0023	0.0014	
No. of test cases	9928	10026	11558	11624	12660	5538	38666	100000	

Version failures which are detected in operational test are listed in Table 6.5. Most of the 34 versions passed all these 100,000 test cases, while only four of them exhibited a number of failures, ranging from 2 to 2760. Compared with the failure data of NASA 4-University project with over 900,000 test cases described in [27], our data demonstrate some similar as well as different reliability features with respect to the same RSDIMU application.

Table 6.6: Coincident failures between versions

Version pairs	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}	Total failures
22 & 34	0	15	0	6	0	4	0	25
29 & 34	0	0	0	0	0	0	32	32
Total no. of failures	0	827	0	653	0	207	3044	4731
Conditional frequency	0	0.0005	0	0.0003	0	0.0006	0.0003	0.0004

In both projects, the programs are generally more reliable for the cases where no sensor failure occurs during operation (i.e., States $S_{0,0}$, $S_{1,0}$, and $S_{2,0}$) than in the situations where one false sensor occurs during operation (i.e., States $S_{0,1}$, $S_{1,1}$, and $S_{2,1}$). This is because there are some complicated computations under the latter situations. Particularly in our project data, no new failure was detected in the former three states, while all the detected failures were revealed under the latter three states and the exceptional states (State S_{others}). Especially for version 29, all the 2760 failures occurred under exceptional system states.

In our operational test, totally 4731 failures were collected for 34 versions, representing an average 139 failures per $100,000$ executions for each version. The failure probability is thus 0.00139 for a single version, with highest probability in state $S_{0,1}$ (with 0.0825). This indicates on average the programs inherit high reliability.

We further investigate the coincident failures between version pairs. Here two or more versions are regarded as correlated if they fail at the same test case, whether their outputs are identical or not. Two correlated version pairs were observed, as listed in Table 6.6.

It can be seen that for version pair 22 & 34, there were 25 coincident failures, thus the percentage of coincident failures versus total failures is 4%

for version 22 and 1.85% for version 34. For the other version pair 29 & 34, 32 coincident failures were observed with the percentage 1.16% for version 29 and 2.37% for version 34. The low probability of coincident failures versus total failures supports the effectiveness of N-version programming, meaning a more reliable system can be expected by this approach from out project data. Moreover, as seen in Table 6.4, there are totally six faults identified in four out of the 34 versions. As noted in [74], the size of these versions varies from 1455 to 4512 source lines of code. We can calculate the average fault density to be roughly one fault per 10,000 lines. This figure is close to industry-standard for high quality software systems.

6.3.2 Reliability Improvement by N-version Programming

To estimate the reliability improvement of N-version programming compared with one single version, we first adopt the simplest statistical method. For 2-version system, there are $C(34, 2) = 561$ combinations out of 34 versions. Considering there are 57 coincident failures observed, the average failure is 57/561=0.102 over $100,000$ executions. In a 3-version system there are $C(34, 3) = 5984$ combinations, so that the average failure probability is $57/5984 = 0.0095$ for 100,000 executions, which implies a 11 times higher reliability than that of a 2-version system. Recall the average failures in single version listed in Table 6.5 is 139; therefore, we obtain the reliability improvement from 3-version system versus one single version of about 15000 (139/0.0095) times. For more accurate comparisons, failure bound models can be exercised to investigate the reliability features for N-version programming compared with one single version. Consequently, we fit the observed operational test failure data to Popov and Strigini failure bound model [14, 86] and estimate the reliability improvement of N-version programming. The estimated failure bounds for version pair (22,34) and (29,34) are listed in Table 6.7.

Table 6.7: Failure bounds for 2-version system

Version pair	DP1		DP2		DP3	
	Lower bound	Upper bound	Lower bound	Upper bound	Lower bound	Upper bound
(22,34)	0.000007	0.000130	0.000342	0.006721	0.000353	0.008396
(29,34)	0.000000	0.000001	0.000009	0.000131	0.000047	0.000654
Average in our project	$1.25 \cdot 10^{-8}$	$2.34 \cdot 10^{-7}$	$6.26 \cdot 10^{-7}$	0.000012	$7.13 \cdot 10^{-7}$	0.000016
Average in NASA project	$2.32 \cdot 10^{-7}$	0.000007	0.000023	0.000103	0.000072	0.000276
Average in NASA project after omitting version 7	$6.15 \cdot 10^{-9}$	0.000006	$2.16 \cdot 10^{-7}$	0.000024	$5.97 \cdot 10^{-7}$	0.000028

Note that DP1, DP2 and DP3 stand for three different testing profiles with various probabilities on different input domains, i.e., $S_{i,j}$ [14]. For simplicity, we denote an insignificant value of the lower bound for version pair (29,34) under DP1 to be zero, which stands for independence between the versions. Since only two version pairs show correlations, we add these bounds up, and then divided by 34, to get the average lower and upper bounds for all 2-version combinations. The average lower bound and upper bounds for any 2-version system under different profiles are shown in Table 6.7. Compared with the average failure probability for single version 0.00139, the reliability improvement of 2-version system versus one single version under DP3 (which is the closest to the real distribution) is 90 to 2,000 times. As the reliability of 3-version system is 11 times of that for 2-version system, the reliability improvement by 3-version system is thus about 900 to 20,000 times over single version system under DP3. Similar improvement can be obtained in DP2, and DP1 achieves even higher improvement.

Moreover, we apply the same failure bound model on the NASA data and get the average failure bounds, as shown in Table 6.7. It is surprise to see that the failure bounds in the NASA project are at least an order of magnitude larger than those in our project. The experimental data in [27] indicate that although there were only seven faults identified in 14 versions, they produced almost 100,000 failures. Particularly, the single fault in version 7 caused more than half of these failures. The fault was caused by incorrect initialization of a variable; however, it was not stated that why the initialization problem was not detected by certification test. We can see that the failure bounds would have been comparable to our project results, had the fault in version 7 been detected and removed in the NASA project.

6.3.3 Comparison with NASA 4-University Project

In NASA 4-University operational test involving over 900,000 test cases, only seven out of 20 versions passed all the test cases. Moreover, a number of these versions, ranging from 2 to 8, were observed to fail simultaneously on the same test cases. In addition, seven related faults were identified which caused the coincident failures. Two of the NASA faults are also related to sensor failure detection and isolation problem. Other faults, e.g., variable initialized incorrectly and failure isolation algorithm implemented in wrong coordinate system, were not observed in our data. In our future research we will investigate failure coincidences between these two projects.

We compare the failure data collected in operational test in both our project and NASA 4-University project, and list some of the reliability related features in Table 6.8. In our experiment, only 2-version coincident failures occurred,

Table 6.8: Quantitative comparison in operational test with NASA 4-University project

Item	Our project	NASA 4-University project
no. of test cases	100,000	920,746
failure probability	0.00139	0.06881
number of faults	6	7
fault density	1 per 10,000 lines	1.8 per 10,000 lines
2-version coincident failures	57	21173
3 or more version coincident failures	0	372
3-version improvement	900 to 20,000 times	80 to 330 times

and no coincident failures among three or more versions were detected. The number of failures and coincident failures in the NASA project is much larger than that in our project, meaning we have achieved a significantly higher reliability figure in our project. Interestingly, the difference on fault number and fault density is not significant for these two projects. Note there is a number of coincident failures in 2- to 8-version combinations in the NASA project, yet the reliability improvement for 3-version system still achieves 80 to 330 times over the single version. Our project obtains similar improvement for 3-version system (i.e., 900 to 20,000 times) over the single version. Considering the average failure rate for a single version is already 50 times better than that in the NASA project, it is understandable that the improvement for 3-version system in our experiment is 30 to 60 times of that in the NASA project.

Overall, from the above comparison between NASA 4-University project and our project, we can derive some variances as well as invariances on N-version programming. The invariances are:

- Both experiments yielded reliable program versions with low failure probability, i.e., 0.06881 and 0.00139 for single version respectively.

- The number of faults identified in the operational test was of similar size, i.e., 7 versus 6 faults.
- The fault density was of similar size, i.e., 1.8 versus 1 fault detected per $10,000$ lines of code.
- Remarkable reliability improvement was obtained for N-version programming in both experiments, i.e., hundreds to tens of thousands of times enhancement.
- Related faults were observed in both projects, in both difficult and easy parts of the application.

Nevertheless, there are some variances between the two projects:

- Some faults identified in the NASA project did not appear in our project, e.g., divide by zero, wrong coordinate system, and incorrect initialization problems.
- More failures were observed in the NASA project than in our project, causing their average failure probability to be an order of magnitude higher than ours.
- In the NASA project, more coincident failures are observed and the failure correlation between versions was more significant, especially among more than three versions. In our project, the fault correlation was reduced, especially if we omit version 34, which contains hard code and poor logic.
- The overall reliability improvement derived from our data is at least an order of magnitude larger than that from the NASA project. This is true for both single version system and N-version system.

The reasons behind the variance and invariance between the two projects can be concluded from the following aspects: First, as the first RSDIMU experiment, NASA 4-University project had to deal with some specification-related faults. Its N-version programming development process became a bit messy, as faults in the specification would be identified and revised, then distributed to the developers for further evaluation. In our project, as we employed a stable version of the specification, such revisions no longer occurred, and programmers could concentrate on the problem solving process. Secondly, there is apparently a significant progress on programming course and training to computer science students over the past 20 yeas. Modern programmers are well disciplined to avoid common programming fault such as divide by zero and uninitialization problems. Lastly, based on the experience accumulated in all the former projects and experiments on N-version programming, we were able to follow a more well-controlled protocol in our experimental procedure. We believe that the N-version programming design and development process can keep further improvement as a vital software reliability engineering technique.

6.4 Discussions

In this experimental evaluation, we perform comprehensive testing and compare the two projects addressing N-version programming. The empirical data show that both similar and dissimilar faults were observed in these two projects. We analyze the reasons behind their similarities and differences. It is evident that the program versions in our project are more reliable than those in the NASA project in the terms of total number of failures and coincident failures revealed in operational test. Our data show the reliability improvement by N-version programming is significantly high, i.e., from 900 to 20,000, over single program versions already with very high reliability. The effectiveness of N-version programming in mission-critical applications is confirmed in our project data.

Moreover, when we examine the faults identified in our project, especially for 22.1, 29.1 and 32.1, we can find that these hard-to-detected faults are only hit by some rare input domains. This means a new strategy should be employed for such faults. As discussed in our previous study [14, 74], code coverage is a good estimator for testing effectiveness. Experimental data show that in our acceptance test, test cases with higher code coverage tend to detect more faults, especially for exceptional test cases. Nevertheless, in this operational test, none of these faults can be detected by the code coverage indicator.

6.5 Summary

In this chapter, we perform an empirical investigation on evaluating reliability features by a comprehensive comparison between two projects. We conduct operational testing involving 100,000 test cases on our 34 program versions and analyze their failures and faults. The data collected in this testing process are compared with those in NASA 4-University project.

Similar as well as dissimilar faults are observed and analyzed, indicating common problems related to the same application in these projects. Less failures are detected in our operational test of our project, including a very small number of coincident failures. This result provides a supportive evidence for N-version programming, and the improvement is attributed to cleaner development protocol, stable specification, experience in N-version programming experiment, and better programmer training.

□ End of chapter.

Chapter 7

Effect of Code Coverage on Fault Detection

In terms of the ability to detect faults, various testing strategies have been evaluated and compared through experiments [32], simulations [26, 40], and analysis [12, 18, 31, 33, 50, 78, 105]. Furthermore, based on the intuition that more faults will be revealed if more code is executed during testing, code coverage has been proposed as an indicator of testing effectiveness and completeness for the purpose of test case selection and evaluation [73, 91, 96]. However, as this remains a controversial issue, more empirical test data with real-world complicated applications are seriously needed to evaluate the effect of code coverage on test case evaluation and selection under various testing strategies.

In this chapter, we will cover the impact of code coverage on fault detection capability, the effects under different testing strategies or profiles, and how code coverage can act as a filter for the design of an effective minimum test set.

7.1 Research Questions

Based on all the previous investigations and evaluations on the effect of code coverage and comparisons of various testing strategies, we focus our empirical study on the following four research questions:

1) Is code coverage a positive indicator for fault detection capability?

2) Does any such effect vary under different testing strategies and profiles?

3) Does any such effect vary with different code coverage metrics?

4) How does code coverage act as a filter to reduce the size of the effective test set?

To address all these research questions, we employ the testing data and results described in Chapter 4 to investigate all the relationships and questions listed above. As we have seen, coverage testing, as well as mutation testing, were employed in our experiment. Furthermore, multiple program versions were developed by different program teams, and each program version spawned dozens of mutants. This extensive investigation yielded a comprehensive set of empirical results, which are described here.

7.2 Effectiveness of Code Coverage

In order to answer the question of whether testing coverage is an effective means of fault detection, we executed the 426 mutants over all test cases and observed whether additional coverage of the code was achieved when the mutants were killed by a new test case. In the experiment, we excluded the mutants which failed upon the first test case (a total of 174 mutants), as we wanted to take a more conservative view in evaluating test coverage by analyzing only those mutants which passed at least the first test case and then failed in later cases. Note that 35 mutants never failed in any of the

1200 test cases. Consequently, there were a total of 217 mutants included in this analysis. In our experiment, each mutant stands for one real fault in the software development process. Thus the terms "fault", "defect", and "mutant" are used interchangeably in the following parts.

Effectiveness of testing coverage in revealing faults is shown in Table 7.1. Here we use the common test coverage measures: block coverage, decision coverage, C-use coverage and P-use coverage [77, 91]. The second to fifth column of Table 7.1 identify the number of faults in relation to changes of blocks, decision, C-uses and P-uses, respectively. For example, "6/8" for version ID "1" under the "Blocks" column means during the evaluation test stage, six out of eight faults in program version 1 showed the property that when these faults were detected by a test case, block coverage of the code increased. On the other hand, two faults of program version 1 were detected by test cases without increasing the block coverage. The last column "Any" counts the total number of mutants whose coverage increased in any of the four coverage measures when the mutants were killed.

The result clearly shows that the increase in coverage is closely related to achieving more fault detections. Out of the 217 mutants under analysis, 155 of them showed some kinds of coverage increase when they were killed. This represents an impressive ratio of 71.4%. The range, however, is very wide (from 33.3% to 90.9%) among different versions. This indicates that the programmer's individual capability accounted for a large degree of variation in the faults they created and the detectability of these faults.

One may hypothesize that when there are more (or fewer) faults in a program version, it may be easier (or more difficult) to detect these faults with coverage-based testing schemes. A plot of the number of mutants against effective percentage of coverage is therefore shown in Figure 7.1. It can be

Table 7.1: Fault detection related to changes of test coverage

Version ID	Blocks	Decisions	C-Use	P-Use	Any
1	6/8	6/8	6/8	7/8	7/8(87.5%)
2	9/14	9/14	9/14	10/14	10/14(71.4%)
3	4/7	4/7	3/7	4/7	4/7(57.1%)
4	7/11	8/11	8/11	8/11	8/11(72.7%)
5	7/10	7/10	5/10	7/10	7/10(70%)
7	5/10	5/10	5/10	5/10	5/10(50%)
8	1/5	2/5	2/5	2/5	2/5(40%)
9	7/9	7/9	7/9	7/9	7/9(77.8%)
12	10/20	17/20	11/20	17/20	18/20(90%)
15	6/11	6/11	6/11	6/11	6/11(54.5%)
17	5/7	5/7	5/7	5/7	5/7(71.4%)
18	5/6	5/6	5/6	5/6	5/6(83.3%)
20	9/11	10/11	8/11	10/11	10/11(90.9%)
22	12/13	12/13	12/13	12/13	12/13(92.3%)
24	5/7	5/7	5/7	5/7	5/7(71.4%)
26	2/12	4/12	4/12	4/12	4/12(33.3%)
27	4/7	5/7	4/7	5/7	5/7(71.4%)
29	10/18	10/18	11/18	10/18	12/18(66.7%)
31	7/11	7/11	7/11	7/11	8/11(72.7%)
32	3/7	4/7	5/7	5/7	5/7(71.4%)
33	7/13	7/13	9/13	10/13	10/13(76.9%)
Overall	131/217 (60.4%)	145/217 (66.8%)	137/217 (63.1%)	152/217 (70%)	155/217 (71.4%)

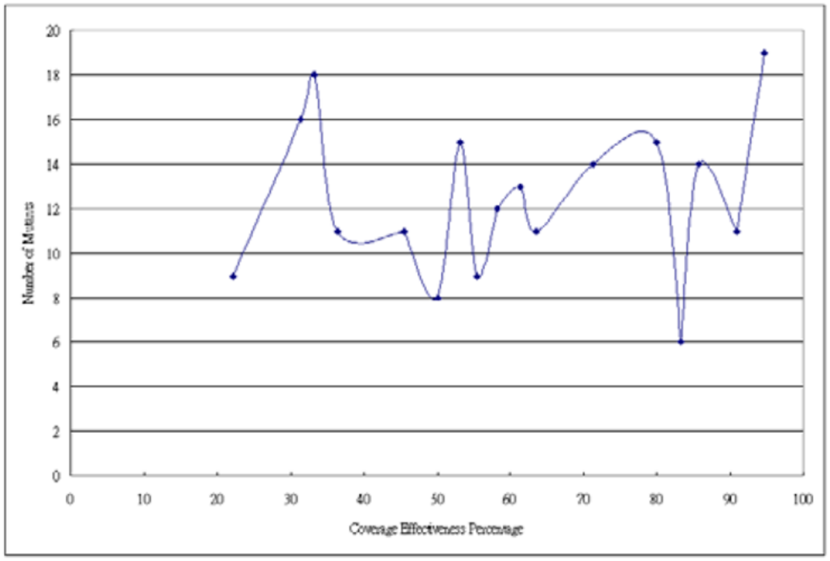

Figure 7.1: Relations between numbers of mutants against effective percentage of coverage

seen that the number of mutants in each version (i.e., the number of faults in the program) can not indicate one way or the other the effectiveness of test coverage in exploring the faults (by killing the mutants).

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 7.2. The vertical axis indicates the average percentage of block coverage by each test case. Lines A, B, C, D, E represent the border for test cases 111, 152, 393, 801 and 1001, respectively. They mark the distinct boundaries of the different test cases described in Table 4.1. Figure 7.2 shows the fault detection capabilities of different kinds of test cases, as separated by the lines. The total average block coverage is 45.86%, with a range from 32.42% to 52.25%.

The decision, C-use and P-use coverage measures expose *exactly* the same pattern except for their absolute values, and are thus omitted here. The overall average value of these measures is shown in Table 7.2.

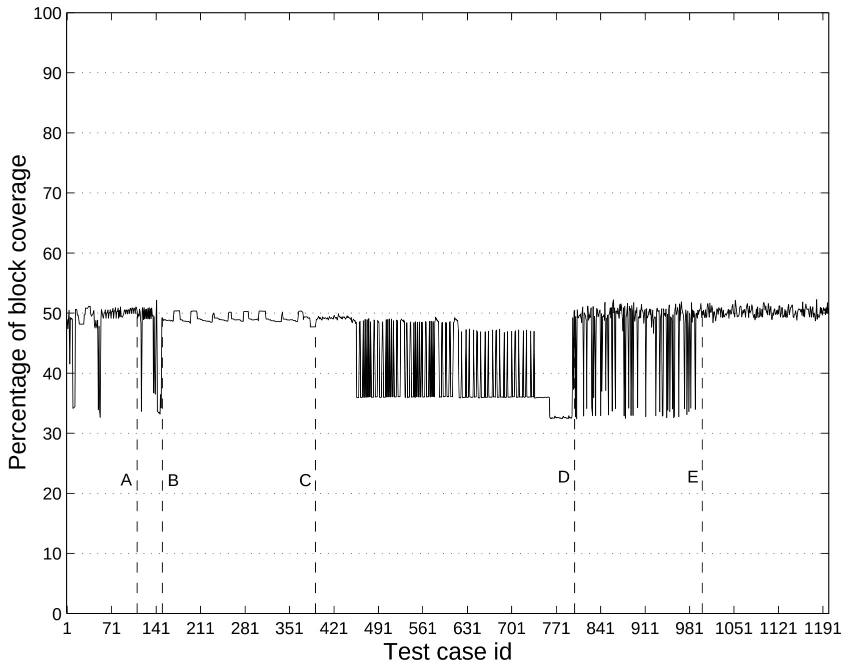

Figure 7.2: Test case contribution on block coverage

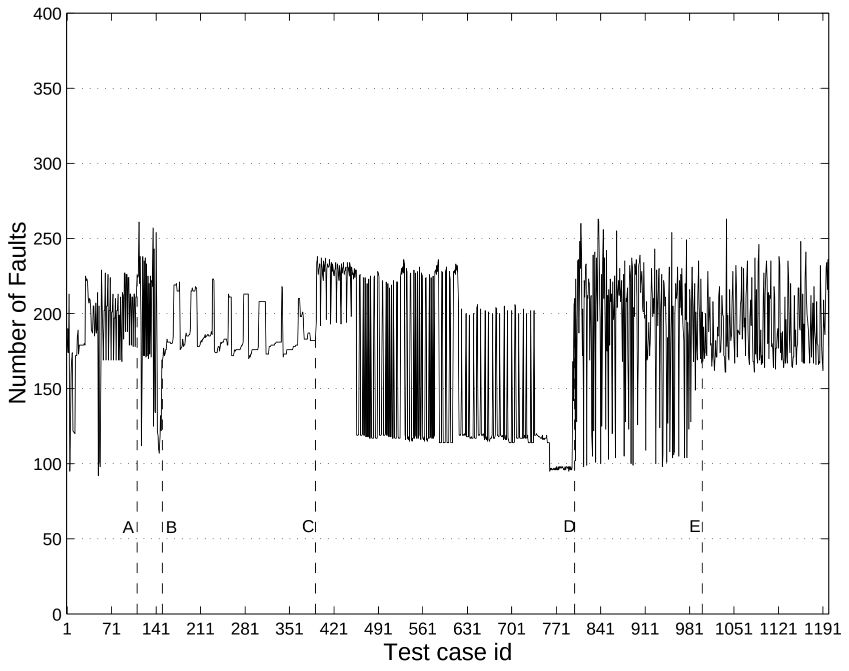

Figure 7.3: Test case contribution on mutant coverage

Table 7.2: Percentage of test case coverage

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

The contribution of each test case in covering (killing) the mutant population is shown in Figure 7.3. The vertical axis represents the number of mutants that could be killed by each test case. Lines A, B, C, D, E represent again the distinct boundaries of different test cases. As in Figure 7.2, Figure 7.3 also clearly portrays the fault detection profiles of each kind of test case. The average number of faults detected by a test case is 172, with 92 as minimum and 263 as maximum.

The comparison between Figure 7.2 and Figure 7.3 offers profound implications: they reveal both the similarity and the difference between *code* coverage and *fault* coverage. On the one hand, test coverage and mutant coverage show similar capability of revealing patterns in the test cases, giving credit to code coverage as a good indicator for test variety. On the other hand, the code coverage value alone is not a good indicator for test quality in terms of fault coverage. Higher and more stable code coverage, e.g., that achieved by test cases 1001-1200, may result in lower and more unstable fault coverage.

Next, if we examine the relationship between cumulated code coverage and defect coverage for all 1200 test cases, we find high correlation between the two, with an R^2 of 0.945, as shown in Figure 7.4. This also demonstrates that code coverage is a positive indicator of fault coverage. The cumulated block coverage and defect coverage according to the 1200-case sequence are shown in Figure 7.5 and Figure 7.6 respectively. In the test case sequence, there are 49 occurrences of new mutants being killed. Out of these 49 occurrences, 38

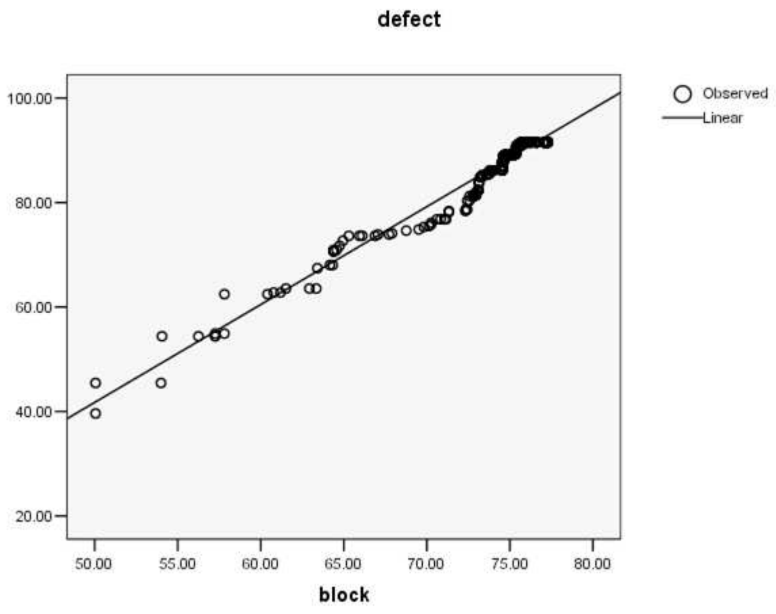

Figure 7.4: Cumulated defect coverage versus block coverage

(77.6%) exhibit some kind of cumulated code coverage increase, while 11 of them (22.4%) show no coverage increase.

Moreover, as shown in Figure 7.2 and Figure 7.3, block coverage and fault coverage show different patterns in different parts of the whole test set. Thus we divide the whole test set into six regions according to their patterns (see Table 7.3). These six clusters also reflect the underlying design principles of different test cases. After applying a linear regression model on the current data, we get the parameters and the quality of fit of the linear models in the various regions as well as in the whole test case space, as illustrated in Table 7.3. The results show that the relationship between block coverage and mutant coverage can be predicted by a linear model over the whole test case space, with an R-squared value of 0.781 (see Figure 7.7). But as a measure of the quality of fit, R^2 ranges dramatically from 0.189 (in Region VI) to 0.98 (in Region IV) in different test case regions, as shown in Figure 7.8 and Figure

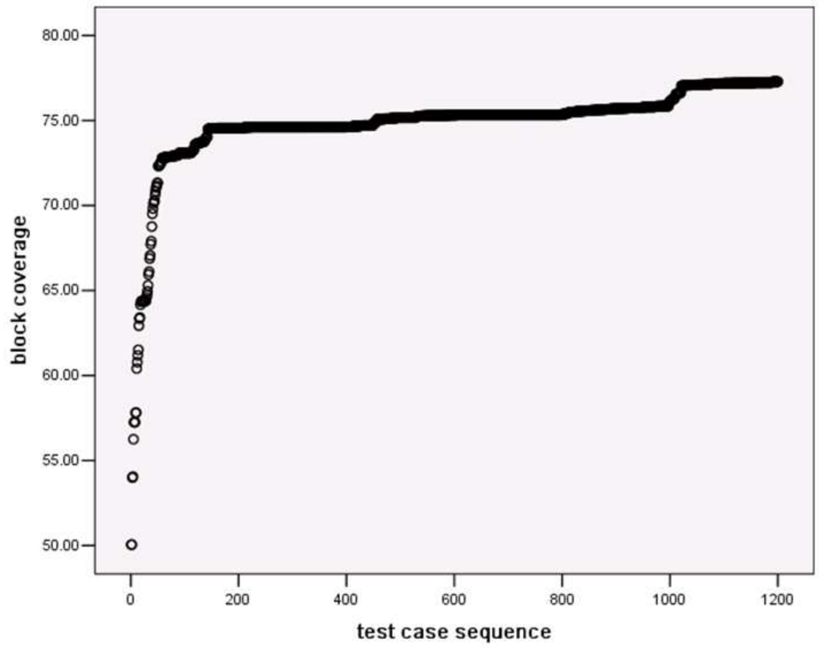

Figure 7.5: Cumulated block coverage in the 1200-case sequence

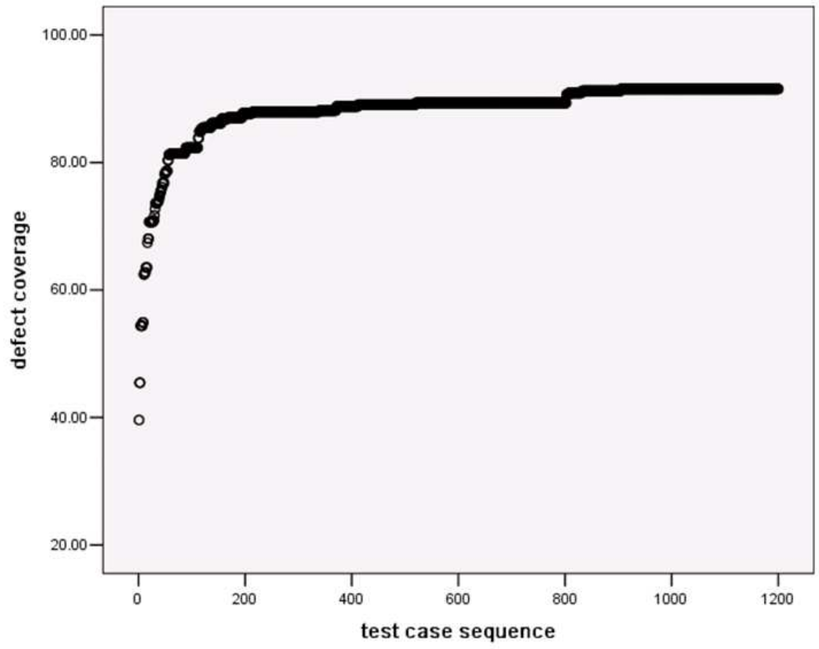

Figure 7.6: Cumulated defect coverage in the 1200-case sequence

Table 7.3: Parameter and fitness of linear models in different test case regions

Test case region	R-squared
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

7.9.

Overall, our experimental data support the hypothesis that code coverage is a positive indicator for fault detection. There are three indications of this: 1) 71.4% of the mutants exhibit some kind of coverage increase when they are killed; 2) the R-squared value of the linear model of block coverage and defect coverage is 0.781 on the whole test set, but it varies from 0.189 (random testing) to 0.981 (one of the functional testing regions); 3) when new faults are detected, 67.3% of the occurrences are associated with an increase in cumulated code coverage on at least one of the four coverage metrics.

In the next section, we consider the differences in code coverage on fault detection between test case regions.

7.3 Effects of Code Coverage under Various Testing Strategies

In the following, we will examine the effect of code coverage on fault detection capability under various testing strategies: 1) various partition subdomains; 2) partition testing versus random testing; and 3) normal operational testing

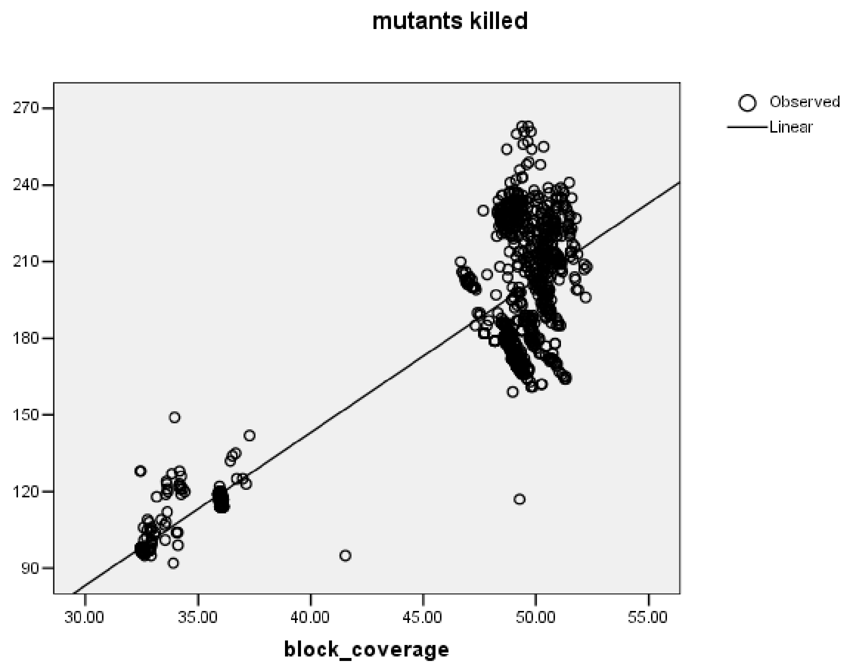

Figure 7.7: Block coverage and defect coverage

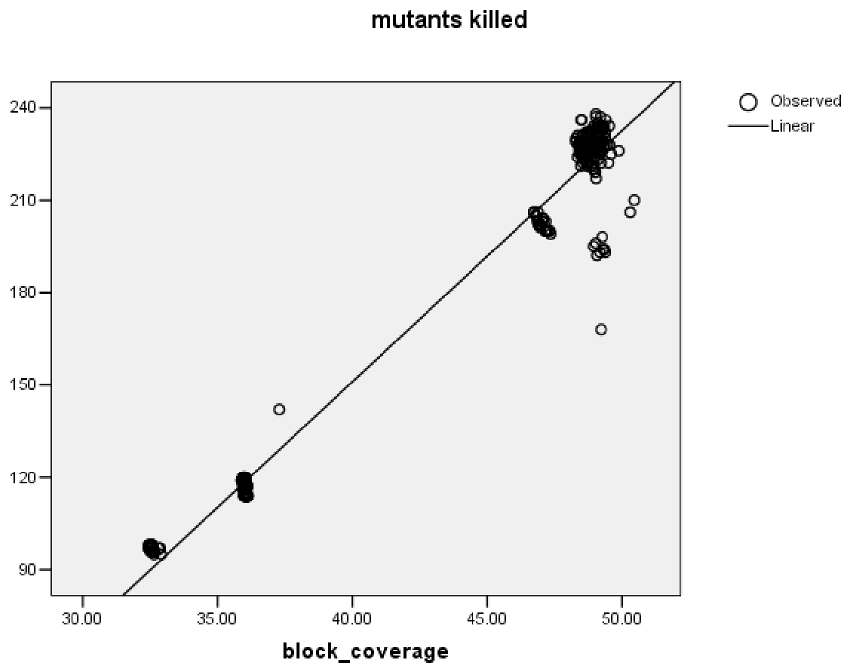

Figure 7.8: Block coverage and defect coverage in region IV

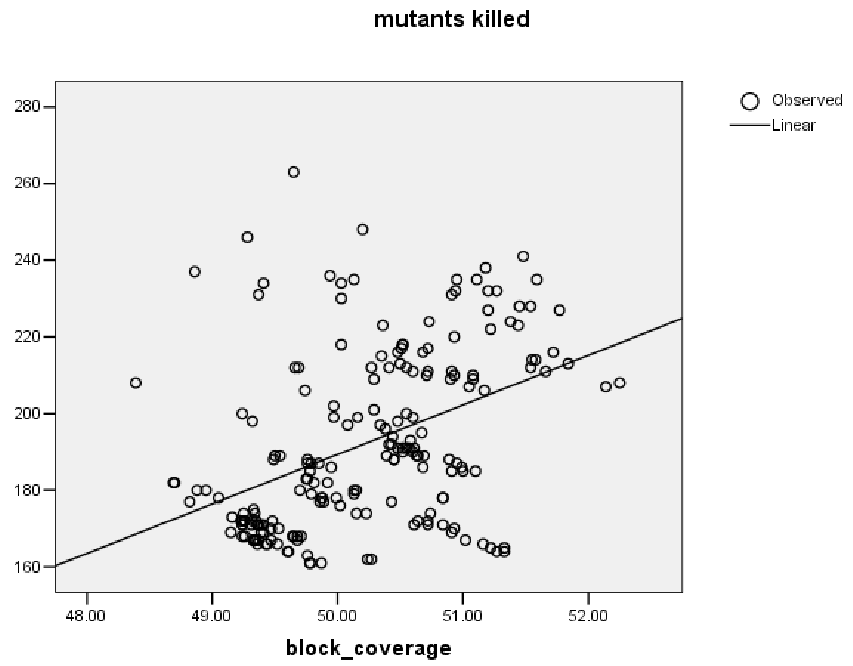

Figure 7.9: Block coverage and defect coverage in region VI

versus exceptional operational testing.

7.3.1 Under Various Partition Subdomains

As mentioned before, the first 800 test cases were designed to target different functions of the system, and the last 400 test cases were randomly generated to simulate the operational environment.

Figure 7.7 indicates that code coverage is a moderate indicator for fault detection capability of the given test cases. Generally, the more code that a test case executes, the more mutants it kills in program versions. But a 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 fewer 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 find the best fit in Region IV and the worst fit in Region VI. Note that the test cases in Region IV are designed with various combinations of the system status, while the test cases in Region VI are randomly generated with a single initial random seed. One of the causes behind this difference in fit quality may be the design principle of test cases in Region IV, which targets the main control flow of the program. The more program code they execute, the more likely it is that program versions fail. This agrees with the traditional assumption and observation that more code coverage brings better fault coverage. The other cause is that, for Region VI, all the 200 test cases have very similar block coverage (from 48% to 52%). This 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 following analysis, we believe the strong correlation in Region IV arises from the fact that large number (277/373) of exceptional test cases are contained in this region.

7.3.2 Partition Testing versus Random Testing

Partition testing and random testing are two basic test data selection criteria. In our test set, 800 test cases are partition 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 7.4: R-squared value in testing profiles

Testing profile (size)	R-squared
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

Table 7.4. The correlation in partition testing is larger than that in random testing, but the difference is not significant. In general, partition test cases are designed to maximize their code coverage (i.e., to cover more code fragments), while random test cases are generated to simulate a real operational environment and are not likely to improve code coverage. From our results, some partition 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 are no exceptional test cases in Region VI, but a large number of exceptional test cases (277 in Region IV out of 373 in total test set) in Region IV. For another random test region, i.e., Region V, a positive correlation is also observed, with $R^2 = 0.778$, as there are 56 exceptional test cases in this region.

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

7.3.3 Normal Operational Testing versus Exceptional Testing

Test cases are designed to detect and remove residual faults in program versions developed to satisfy the requirements in the software specification. There are two major system statuses, 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. In this RSDIMU application, normal operation refers to those situations where at most two sensors fail during the input and at most one sensor fails during the test. All other cases, which cause difficult conditions such that acceleration of the vehicle cannot be estimated, are viewed as exceptional operations.

As shown in Table 7.4, the linear correlation of code coverage and fault coverage changes dramatically from normal testing (0.045) to exceptional testing (0.944). This is shown in Figure 7.10 and Figure 7.11, respectively.

This may be explained as follows. In normal testing, the code coverage range is relatively small (see Figure 7.10), 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 higher code coverage may be obtained (compared with that of exceptional testing), it cannot be employed to predict the fault detection capability of a normal operational test case. Normal test cases with similar code coverage can be classified into different input/output subdomains, some of which may be more difficult than others. The program fragments which target more difficult subdomains are more fault-prone, while the fragments with easier subdomains are more reliable. So besides code coverage, the difficulty of subdomains may

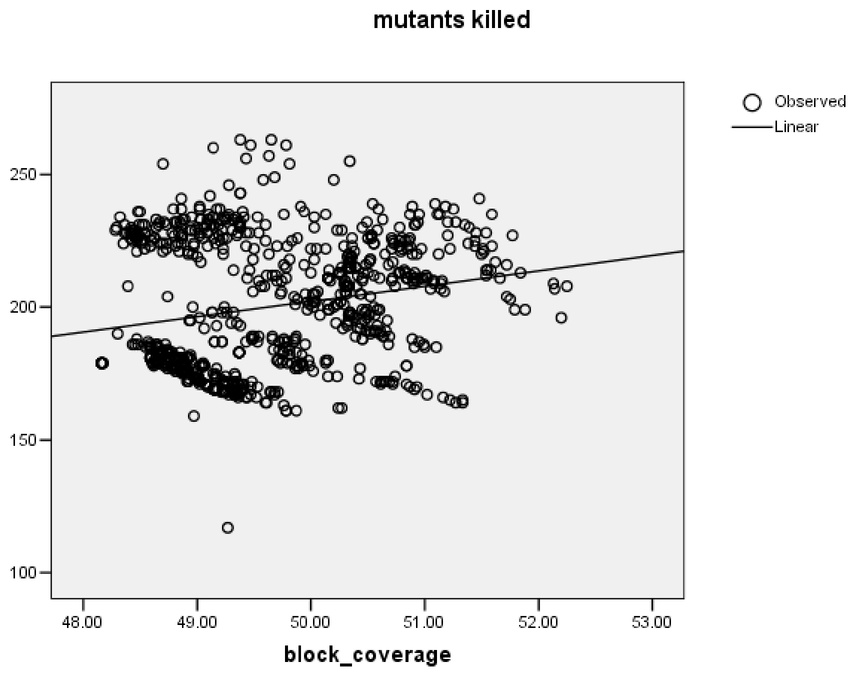

Figure 7.10: Block coverage and defect coverage in normal testing

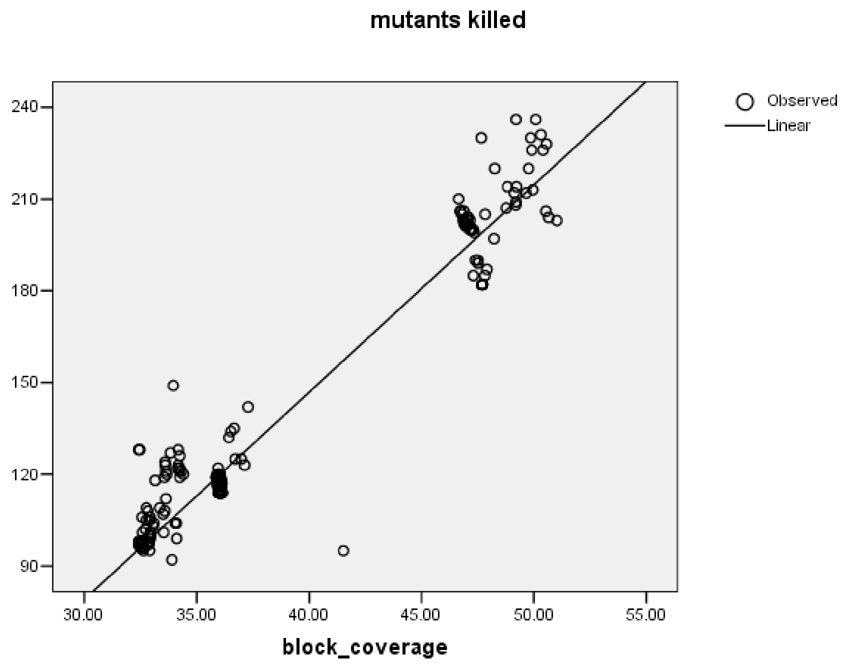

Figure 7.11: Block coverage and defect coverage in exceptional testing

be another factor of fault coverage.

Figure 7.11 contains two main clusters. We examine the exceptional test cases and find that these two clusters 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 can be skipped (e.g., when four sensors on exactly two faces have failed before the test, and no additional sensor fails during the test); in other cases, all this computational code is skipped, as determined by the system status. This explains why the code coverage shows two different ranges, and why a large gap exists between the two clusters. Although this phenomenon is application-specific, the strong correlation pattern provides positive support for the use of code coverage as a measure of fault 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 than in 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.

7.4 Combinations of Various Coverage Metrics

From the data shown above, we observe that the effect of code coverage on fault coverage is significant in exceptional testing, but weak in normal testing. The reason for the difference between functional testing and random testing

Table 7.5: Linear regression fitness for combinations

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

Table 7.6: R-squared 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

is not obvious, but nonetheless code coverage is a moderate indicator for test effectiveness. To further illustrate this effect, we examine the correlation pattern in different testing profile combinations. The linear regression fits in the four combinations are listed in Table 7.5. It is clear 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 a weak correlation.

To investigate the correlation pattern between different code coverage metrics and test effectiveness under various testing profiles, the R-squared values of linear regression for decision coverage, C-use and P-use are listed in Table 7.6, compared with that for block coverage. The other three coverage metrics show similar patterns to 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 observations described above, larger variation in code coverage implies more correlation in terms of the relationship among different clusters.

To further investigate the prediction performance of coverage metrics, we look at the different coverage increase when new mutants are killed. As stated earlier, there are 49 occurrences of new mutants being killed in the whole test set, while 33 of them exhibit some kinds of cumulated coverage increase. Out of these 33 occurrences, three of them do not show block coverage increase, which implies an inaccuracy of 8.3% with block coverage. Nevertheless, this figure is not significantly different to other coverage metrics.

Overall, our experimental data show that no significant prediction performance difference between block coverage, decision coverage, C-use and P-use.

7.5 Reduction of the Size of the Effective Test Set

Test set minimization aims to reduce the cost associated with re-testing [108]. The problem is equivalent to the NP-complete *minimal set covering problem* [35]. An implicit enumeration algorithm with reductions has been employed to find the optimal set covering [108], showing that the time cost may be up to exponential. Several greedy or heuristic methods have been proposed and evaluated to obtain approximate solutions [41, 111]. Moreover, tradeoffs between test set reduction and fault detection capability are also under investigations [6].

Here we employ a simple method to obtain the effective test set using coverage increase as a filter. The sizes of the test sets using different coverage increase criteria on the whole test set are listed in Table 7.7. The original test set with 1200 test cases can kill 391 out of 426 mutants (although 35 mutants contain real faults, they cannot be detected with any of the 1200 test cases, reflecting the fact that testing alone is not enough for ensuring software reliability). If each test case with 0.01% increase in at least one of the four coverage measurements (block, decision, C-use and P-use) is selected, the size of final test set is 203, which is only 16.9% of the original set; nevertheless it can kill 374 mutants, which is 95.7% of the faults detected by the original test set. Then we randomly select 203 test cases from the original 1200, and investigate the number of mutants killed by it. The random selection is repeated 100 times, and find that the average number of mutants detected by these 100 test sets is 371, which is listed in the last column.

Table 7.7: Reduction of the size of test set with coverage increase

Criteria	test set size (%)	mutants killed (%)	mutants killed by random set
original	1200(100%)	391(100%)	—
block inc.>0.01%	116 (9.7%)	359 (91.8%)	363
block inc.>0.05%	78 (6.5%)	356 (91.0%)	358
block inc.>0.25%	40 (3.3%)	349 (89.3%)	346
block inc.>1%	18 (1.5%)	328 (83.9%)	323
block inc.>2%	11 (0.9%)	317 (81.1%)	302

Table 7.7 agrees with the intuitive that the test set size will be smaller if the coverage increase criterion is larger, although the fault detection capability reduces as well. As illustrated in Figure 7.3, the average number of faults detected by a test case is 178, with 92 as minimum and 263 as maximum. Since the differences between the mutants killed by reduced test set and those by random set are insignificant, as shown in Table 7.7, the data is not evident to answer the question whether coverage increase is helpful or not for test set

Table 7.8: Test set reduction with normal testing

Criteria	test set size	mutants killed	mutants killed by random set
original	827	371	—
block inc.>0.01%	87	351	353 (100.6%)
block inc.>0.05%	59	346	348 (100.6%)
block inc.> 0.25%	28	341	334 (97.9%)
block inc.>1%	11	308	304 (98.7%)
block inc.>2%	8	303	292 (96.4%)

Table 7.9: Test set reduction with exceptional testing

Criteria	test set size	mutants killed	mutants killed by random set
original	373	355	—
block inc.>0.01%	29	327	298 (91.1%)
block inc.>0.05%	19	316	277 (87.7%)
block inc.>0.25%	12	270	243 (90.0%)
block inc.>1%	7	238	216 (90.8%)
block inc.>2%	3	228	180 (78.9%)

reduction.

To further investigate the difference in performance under normal or exceptional test cases, we break up the reduced test set into normal test cases and exceptional test cases, and select random test sets accordingly. The numbers are listed in Table 7.8 and Table 7.9 respectively. For example, the test set of size of 116 (when using block increase greater than 0.01% as a filter) contains 87 normal test cases and 29 exceptional test cases. So in Table 7.8, 87 normal test cases are selected randomly from the original 827 normal test cases, and find that the average number of faults detected by 100 such randomly-generated test sets is 353, slightly more (100.6%) than that of the reduced test set which is 351. Next in Table 7.9, when we examine the 100 test sets which contains 29 random exceptional test cases, 298 mutants are killed on average.

This shows fault detection capability of 91.1% of the reduced test set (327 mutants killed).

The following observations can be drawn from Tables 7.7, 7.8 and 7.9:

1. For the whole test set, when coverage increase (whether small or large) is used as the condition to select the reduced test set, the number of faults detected is almost the same as that of a random test set of the same size; thus, we can not conclude coverage is an effective indicator for test set reduction.

2. For normal test cases, the performance of the random test set is always similar to that of the coverage-increase test set, except in the case with the largest coverage increase (where the number of the faults detected by the random set is 292 versus 303, i.e., 96.4%). So better code coverage does not imply a sound testing performance for normal test cases.

3. For exceptional test cases, the performance of the reduced test set selected by coverage increase is always better than that of the random test set. The number of faults detected by the latter ranges from 78.9% to 92.2% of those found by the former. Hence, code coverage is a good filter for reducing the size of the test set with the smallest concomitant reduction of the testing effectiveness, especially when the coverage increase margin is larger, say, 2%.

These observations further verify our finding in the previous sections that code coverage is a helpful criterion for the design and evaluation of exceptional test cases.

7.6 Assessment of Various Testing Strategies

7.6.1 Comparisons of Partition Testing and Random Testing

The effectiveness of random testing has been a controversial issue for some time [107]. As to whether random testing is 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 to that of functional test cases, e.g., the similar code coverage (around 50%) obtained in Region VI compared with that in Region IV. Secondly, random testing can kill mutants whose faults are hard to detect, i.e., with a small rate of failure occurrence. If we examine the failure details of mutants that failed in fewer than 20 test cases (which means these mutants inherit a low failure occurrence), we find that there are 94 random test cases and 169 functional test cases that can detect these faults. Considering the ratio of random test case to the whole test set is 33.3% ($\frac{400}{1200}$), the percentage 35.7% ($\frac{94}{94+169}$) shows that random test cases are as effective in detecting hard-to-kill mutants as functional test cases.

The numbers and failure occurrence of mutants that failed in functional testing only or in random testing only are listed in Table 7.10. 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 in both modes of testing, 20 mutants (with mean failure number of 4.5) were killed by functional testing only and nine mutants (with 3.67 failures in average) failed in random test cases only. This means that random testing may miss 5.2% (20/382) of faults compared with functional testing, but it kills 2.4% (9/371) additional faults which are not detected by functional testing. These

Table 7.10: The number of mutants failing 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

nine newly-killed mutants inherit rather low failure occurrence.

Overall, these results show that random testing is a necessary complement to partition testing.

7.6.2 Normal Function Testing versus Exceptional Testing

According to Table 7.10, the mutants killed by exceptional testing only fail less frequently (with 55 failures on average) than those failing under normal testing only (with 120 failures in average). Considering that 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 7.10 also reveals that mean failure numbers under functional testing and random testing are significantly different from those under normal testing and exceptional testing. This may reflect the different features and relationships among the four testing profiles. Functional testing (which is designed according to the specification) and random testing (which is designed according to operational profile) have a considerable overlap. Most cases under the two testing profiles can detect similar faults. Only a small number of function-specific faults or faults occurring under extreme situations can be detected by functional testing or random testing only. In the meanwhile, the occurrences of the test cases that can detect such faults are relatively fewer. In contrast, the profiles of normal testing and exceptional testing are parallel, i.e., they contain no overlap. A fault occurring only under normal operations may fail in many normal test cases, but it cannot be detected by exceptional testing, and vice versa. At the same time, the number of normal or exceptional test cases that can detect such faults is larger since these test cases may aim at the same subdomains. The different features and relationships between 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. Code coverage is clearly a good indicator of fault detection capability of exceptional test cases, but not of normal test cases. This can give some hints on designing the exceptional test cases: increasing the code coverage of such test cases will yield better fault detection capability.

7.7 Detailed Analysis in Region IV

As shown in Figure 7.8, a high correlation between code coverage and defect coverage was observed in Region IV. Moreover, as exceptional testing exhibits a higher correlation than normal testing, the exceptional test cases in Region IV should give us more hints about subdomain-based exceptional testing. The block coverage and defect coverage of all the 277 exceptional test cases in

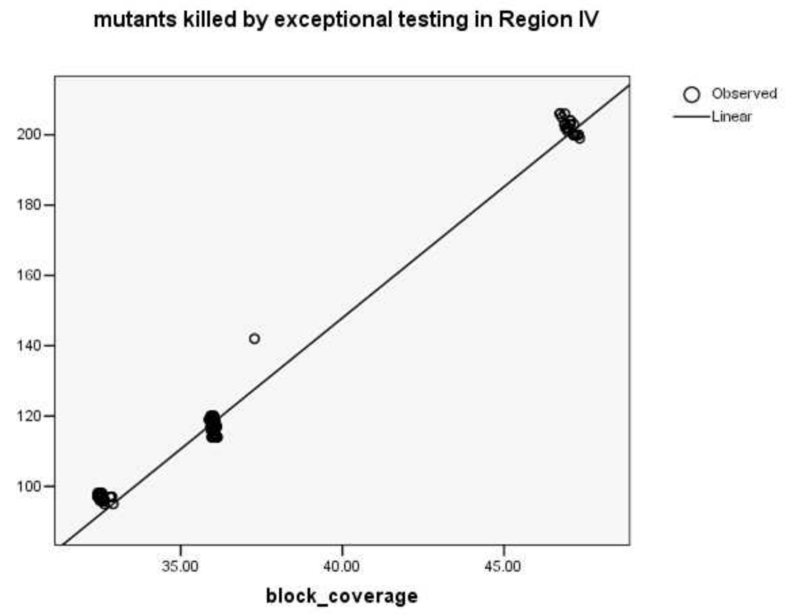

Figure 7.12: Block coverage and defect coverage in exceptional test of region IV

Region IV are illustrated in Figure 7.12. Clearly, there are three main clusters, and a single point which is away from the best-fit line.

In the lowest cluster, which exhibits smallest block coverage and defect coverage, the 36 test cases belong to two different subdomains: these represent the failure of either seven or six sensors in the inputs. In this cluster, the range of block coverage is from 30% to 35%, and the range of defect coverage is from 90 to 100 mutants.

In the middle cluster, the 216 test cases can be classified into three subdomains: two, three or four sensors failing in the input, respectively. The block coverage is in the range of 35% to 40%, and the number of mutants killed is around 120.

The third cluster, which represents the highest coverage of both code and defects, contains 24 exceptional test cases from one single subdomain: three sensors failed before the test, but as two of them are on the same face and all the other sensors are operational, only two faces are actually completely operational, but the system status is still operational and the estimation can still be made. According to the specification, the system status is analytic; this is a extreme boundary for the normal operational case, in which the vehicle state can be estimated while the channels' state cannot be estimated as the available information is limited. Thus this subdomain is regarded as an exceptional subdomain in the literature [28, 86]. In this case, 45% to 50% block coverage is covered, and about 200 mutants can be detected.

The only point that lies outside the line is from another subdomain, where no sensor failed as the input, but three of them were regarded giving only noise during calibration. As a result, all the sensors are regarded as having failed.

In summary, the high correlation of exceptional testing in Region IV reveals that the increase of code coverage in subdomain-based testing does affect the fault detection capability. Moreover, this testing strategy itself can increase such capability, as can be seen from our data.

7.8 Discussions

7.8.1 Threats to Validity

The discussions to general threats to validity of the whole project can be found in Chapter 4. Here we just highlight some possible threats to validity related to coverage data collection.

Although there are many other coverage measurements available, we believe these four metrics can adequately represent the basic data-flow and control-flow coverage measurements. Actually, when we examine the various relationships between these metrics, the patterns are similar to each other. This phenomenon has also been observed in previous studies [15, 76]. So we choose block coverage as the representative code coverage measurement in this study, although all the relationships can be obtained for the other three measurements.

For determining the testing effectiveness, the design strategy of the test set becomes critical. In this experiment, the first 800 test cases were carefully defined by the domain experts to address the requirements of the specification. The test set has been adopted as an acceptance test for some previous empirical studies.

In our investigation of the effect of code coverage, each mutant is treated as a single fault. The code coverage of the mutants is regarded as equal to the code coverage of the final version after removing the corresponding fault. Although there is a slight discrepancy between the two degrees of coverage, we think the difference may not be very significant because, on average, each fault affects only a few lines of code.

7.8.2 Implications of Our Results

Based on our project data, we investigate the effect of different code coverage metrics under different testing profiles. We focus on the following four questions: 1) Is code coverage a positive indicator for fault detection capability? 2) Does this relationship vary under different testing profiles? 3) Does it vary with different code coverage metrics? 4) How well does code coverage act as a filter to reduce the size of the effective test set?

For the first question, based on the above experimental data, our answer is positive. Our empirical data show that, in most situations, 71.4% in fact, there is an increase with code coverage when a test case detect additional new faults. Furthermore, in some functional and exceptional testing regions (e.g., region IV), the correlation between code coverage and fault detection is very high, which indicates that high code coverage brings high fault coverage under an exceptional testing profile.

However, we find that the correlation varies under different testing profiles, which answers the second question. As mentioned above, there is a significant correlation between code coverage and fault detection capability for exceptional test cases. A positive linear correlation holds with an overall R-squared of 0.944. In contrast, there is no such correlation for normal operational test cases. The phenomenon of different correlations appearing in different test case regions can be explained by the ratio of exceptional test cases in these regions. Since high correlation holds for exceptional test cases, the large number (277/373) of exceptional test cases contained in Region IV leads to the strong positive correlation in this region.

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

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

As our project data are based on the RSDIMU application, which is computation intensive, the size of some functions is very large compared with those in 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. This is the reason behind the two clusters shown in some of the patterns. As RSDIMU is a real-world critical application from the 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 single case study, further real-world empirical data are still needed.

For the last question, our experimental data show that code coverage is a good filter which can help to greatly reduce the size of the effective test set. Significantly, there is only a small associated decrease in the fault detection capability. In particular, coverage increase information is very useful for reducing the size of the exceptional test cases in the effective test set. The result shows that the performance of the reduced exceptional test set using coverage information is always better, on average, than that of a random test set of the same size. This finding will help to guide the design of the effective test set with an acceptable size, especially for exceptional test cases.

Overall, we can see that using code coverage information to reduce the test set size is more effective for exceptional test cases than normal test cases, according to our experimental data. One possible reason may be that, for normal test cases, they cover the more complicated part of the source code, where faults are detected not only by being executed at least once, but also by examining the values of different variables and the various control flow branches. However, for exceptional test cases, most of the code they cover is logically simpler, and with less chance of being executed. Therefore, using code coverage performs better in exceptional test cases than in normal test cases.

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.

As one possible further research direction, a new testing strategy which combines the concept of domain testing and coverage testing may give more realistic and effective guidance for software testing.

7.9 Summary

In this chapter, we investigate the effect of code coverage on fault detection under different testing profiles, using different coverage metrics, and study its application in reducing test set size.

A unique contribution of our work is an innovative investigation on the relationship between code coverage and fault detection in terms of 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, but weak in normal testing.

Furthermore, there is little evidence for variation between different coverage metrics. All the four coverage metrics studied show similar patterns in the linear relationship between code coverage and fault detection. Moreover, the data support the effectiveness of random test cases due to their significant fault detection capability.

Our study also shows that code coverage can be used as a good filter to reduce the size of the effective test set, although it is more powerful for exceptional test cases. This reinforces the evidence that code coverage is a good indicator, and can be usefully applied in test case design and evaluation, especially for subdomain-based exceptional testing.

In summary, 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.

□ **End of chapter.**

Chapter 8

Predicting Reliability With Code Coverage

As the key factor in software quality, software reliability quantifies software failures. Defined as the probability that a software system does not fail in a specified period of time in a specified environment, software reliability has become the most an essential ingredient in customer satisfaction [70]. As a result, many analytical models have been proposed for software reliability estimation. The time-domain models, also called software reliability growth models (SRGM), have been drawn most attention. These software reliability models use the failures collected in testing phases to predict the failure occurrences in the operational environment. There are two classes of basic data used in traditional SRGMs: 1) failures per time period; or 2) time between failures. A number of reliability models have been proposed to illustrate various distributions between failure/time and reliability, including some well-known models, e.g., Goel-Okumoto and Musa-Okumoto models [70].

Although some of the historical SRGMs have been widely adopted to predict software reliability, researchers believe they can further improve the prediction accuracy of these models by adding other important factors which affect the final software quality [15, 76, 102]. As we have discussed in previous chapters, code coverage that a test set achieves may have certain effect on the reliability of the software system under test. To incorporate the effect of code coverage on reliability to traditional software reliability models, [15] proposes a technique using both time and code coverage measurement for reliability prediction. Basically, it reduces the execution time by a parameterized factor when the test case neither increase code coverage nor causes a failure. Experiments show that the adjusted G-O and M-O models with such time reduction achieve more accurate predictions than the original ones.

In this chapter, we propose a novel method to integrate time and code coverage measurements together to predict the reliability. Before that, we first formulate the relationship between the number of failures and the code coverage achieved by test cases with two simplified models. The key idea of the new reliability model is that the reliability of a software system is not only affected by the execution time it experiences, but also the completeness of the testing. Thus the reliability prediction is composed of two parts: the estimation from execution time and from test coverage. For the effect of coverage on reliability, we propose two models to describe such relationships. For the effect of time on reliability, distributions from traditional SRGMs can be adopted for the estimation.

In literature, several models have been proposed to formulate the relationship between the number of failures/faults and test coverage achieved, with various distributions. [102] suggests that this relation is a variant of Rayleigh distribution, while [76] derives that it can be expressed as a logarithmic-exponential formula, based on the assumption that both defect coverage and test coverage follow Musa-Okumoto logarithmic growth model with respect to execution time.

In the following, we formulate such relation from two different aspects of views. Experimental results will be provided to evaluate the estimation accuracy. Furthermore, we propose a new software reliability model to estimate software reliability with both execution time and test coverage information as two major factors.

8.1 Two Models on Defect Coverage and Test Coverage

8.1.1 A Hyperexponential Model

According to our previous observations, the relationship between the number of faults detected and test coverage achieved varies under different testing strategies. Based on this, we have the following assumptions:

1. There are K classes on the whole test set, showing the different natures of various testing strategies;
2. Within each class, the fault detection rate with respect to coverage is proportional to the number of faults remaining undetected;
3. A fault is corrected instantaneously without introducing new faults;

Following these assumptions, the fault detection rate with respect to coverage within each class is:

$$\frac{dF_c}{dc} = \beta \cdot F_r = \beta \cdot (N - F_c)$$

where F_c is the current cumulated number of faults detected when coverage c is achieved, F_r is the residual faults, N is the total number of faults that are detectable by the current testing strategy, and β is a constant.

Solution of this differential equation in the range of $0 \leq c \leq 1$, under initial condition $F_c = 0$, is as the following:

$$F_c = N(1 - e^{-\beta c}) \tag{8.1}$$

From the assumptions, each class follows the nonhomogeneous Poisson process (NHPP) model with its own parameters [70]. So on the whole test set, the expected cumulated number of faults detected when coverage c is:

$$F_c = \sum_{i=1}^{K} N_i(1 - e^{-\beta_i c}) \tag{8.2}$$

Notice that if $K = 1$ we have the NHPP model. Moreover, as N_i represents the expected total number of faults to be eventually detected in each class, the summation $\sum_{i=1}^{K} N_i$ is the total number of faults that will be detected under various testing strategies.

Since the failure intensity function is the derivative of F_c, we therefore have

$$\lambda(c) = \sum_{i=1}^{K} N_i \beta_i e^{-\beta_i c} \tag{8.3}$$

The two parameters in each class can be estimated using maximum likelihood estimation (MLE) method or least-squares estimation (LSE), using the failure data and coverage information under that particular testing strategy.

8.1.2 A Beta Model

Unlike the hyperexponential model presented above, following the well-known Goel and Okumoto reliability growth model, we assume that both fault coverage and test coverage follow the NHPP model with respect to execution time, i.e.:

$$F_c(t) = N_1(1 - e^{-b_1 t}) \tag{8.4}$$

where $F_c(t)$ is the number of cumulated faults detected at time t, N_1 is the expected number of faults detected eventually, and b_1 is a constant.

Similarly, we have

$$c(t) = N_2(1 - e^{-b_2 t}) \tag{8.5}$$

where c(t) is the cumulated test coverage achieved at time t, N_2 is the ultimate test coverage that can be achieved by testing, and b_1 is a constant.

From (8.5), we can derive the formula for time t,

$$t = -\frac{1}{b_1} log(1 - \frac{c}{N2})$$

Substitute t in (8.4), we get

$$F_c = N_1[1 - (1 - \frac{c}{N_2})^{\alpha}] \tag{8.6}$$

where $\alpha = b_1/b_2$.

From (8.6), the relationship between fault coverage and test coverage follows a Beta distribution, where $\frac{c}{N_2} < 1$. Similarly, the parameters N_1, N_2 and α can be estimated by MLE or LSE methods using fault and coverage data collected during our acceptance test and operational test.

Table 8.1: Estimated Parameters in Hyper-exponential model

Modeling	N	β	SSE
NHPP (1)	1475	0.39	146110
NHPP (2)	5467	0.096	118200
Hyper-exponential	4087	–	23928
Region I	1989	0.256	22195
Region II	476	1.97	133
Region III	411	3.29	1315
Region IV	406	3.75	66
Region V	414	3.77	219
Region VI	391	21.3	1.01e-009

8.1.3 Empirical Evaluation

We apply the failure and coverage data collected in the acceptance test of our experiment to evaluate the two simplified models above. The LSE method is used to estimate the parameters in the models.

First of all, we evaluate the NHPP model as well as the hyper-exponential model. The estimated parameters and the sum of squared errors (SSE) are listed in Table 8.1. It shows that the NHPP model does not fit the failure/coverage data very well, although the SSE will be slightly smaller if N keeps increasing and β keeping decreasing. If hyper-exponential modeling is applied on the six testing regions as shown in Table 7.3, it can be noted that the SSEs of Region II to VI are pretty small. This agrees with the fact that the underlying design strategies of various test cases illustrated in Table 4.1. For the reason why Region I exhibits such a high diversity, we can also find the answer in Table 4.1 since Region I combines all the test cases which target at some basic functions in the programs.

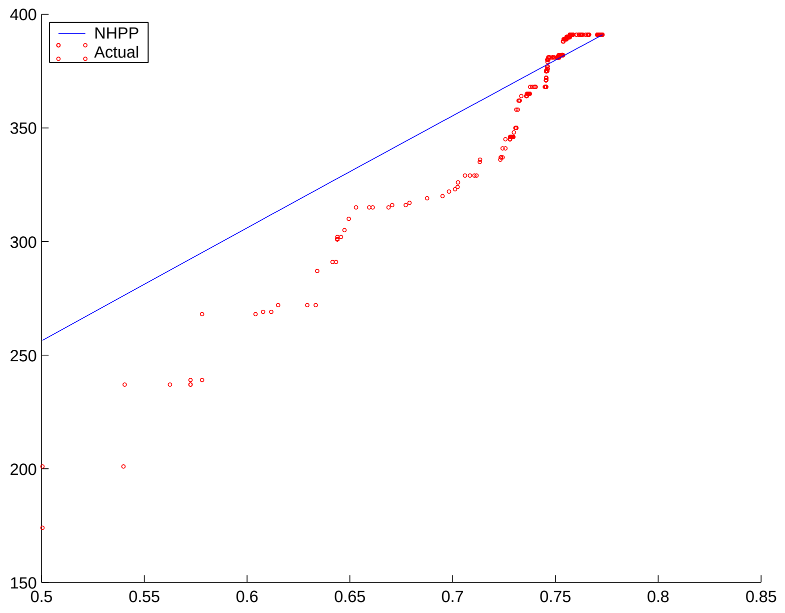

Figure 8.1: NHPP estimation

The NHPP model and hyper-exponential model curves are shown in Figure 8.1 and Figure 8.2 respectively.

For the Beta model, if we assume the ultimate test coverage N_2 in (8.6) is 100%, we can get

$$F_c = N_1[1 - (1 - c)^\alpha] \tag{8.7}$$

Similarly, using LSE method, we can derive the following Beta model:

$$F_c = 1101 \times [1 - (1 - c)^{0.303}]$$

The estimation result is shown in Figure 8.3. The SSE in this estimation is 38365, which is smaller than the NHPP model, but larger than the hyper-exponential model. The comparison of the Hyper-exponential and Beta modeling is shown in Figure 8.4.

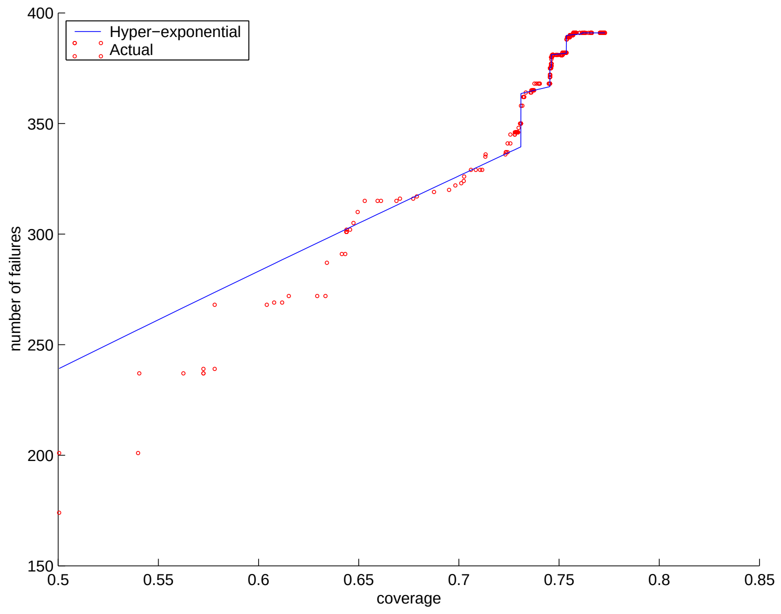

Figure 8.2: Hyper-exponential estimation

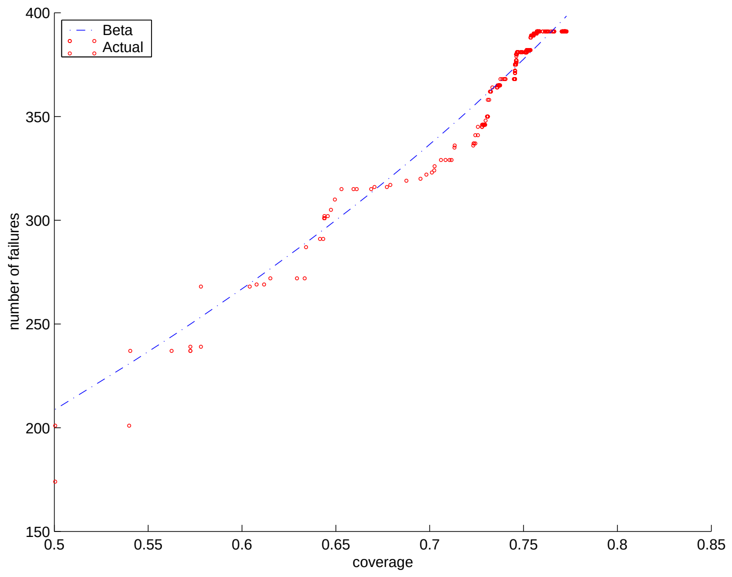

Figure 8.3: Beta estimation

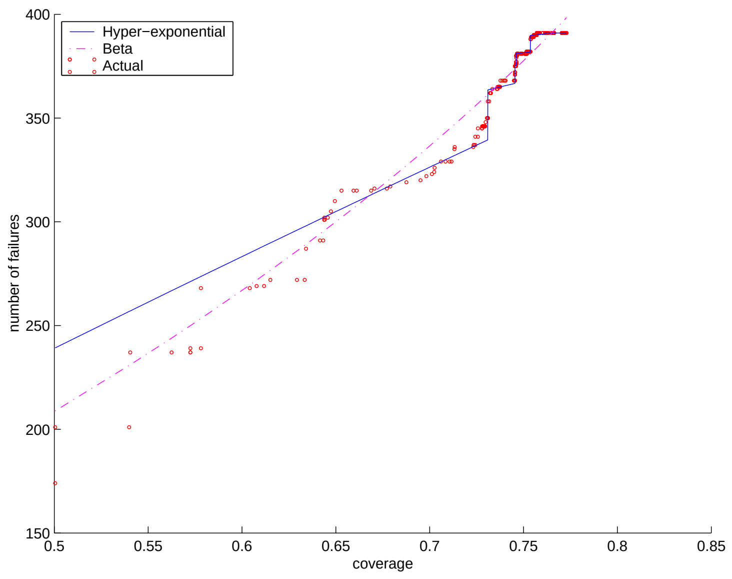

Figure 8.4: Comparison of Hyper-exponential and Beta estimation

8.2 A New Software Reliability Model

Most of the software reliability models are based on time domain, i.e., using either the elapsed time between software failures or the number of failures occurring over a specified time period [70]. However, when we examine the testing procedure, we find that execution time should not be the only factor that affects the failure behavior of the software. For example, if testing sequence within a test set is changed, the time between failures or the number of failures within a certain time period may be different. Under such situation, although the same test set are executed on the same software system, different reliability predictions are made according to the traditional time-domain reliability growth models.

As have been mentioned above, [15] proposes a technique using both time and code coverage measurement for reliability prediction. It is based on the idea that the execution time between failures can be reduced when the test case neither increase code coverage nor causes a failure. Using such a reduction factor, the original G-O and M-O models can be adjusted to achieve more accurate predictions. Overall, [15] propose a method dealing with the adjustment of the time parameter in existing reliability growth models using coverage information.

Here we propose a new reliability model which aims to predict the reliability performance using time between failures and coverage measurement together. The detailed assumptions and model form are illustrated as the following.

8.2.1 Assumptions

Our new reliability model is based on the following assumptions:

1. The number of failures revealed in testing is related to not only the execution time, but also the code coverage achieved by the current test set;
2. The failure rate with respect to time and test coverage together is a parameterized summation of those with respect to time or coverage alone;
3. The probabilities of failure with respect to time and coverage are not independent, they affect each other by an exponential rate.

According to these assumptions, the data requirements to implement this model are: the time between failures, or the actual sequences of test cases that the software failed, and the cumulated coverage measurement achieved by the whole test set.

8.2.2 Model Form

From the assumptions above, we can derive the joint failure intensity function with respect to both time and coverage as following:

$$\lambda(t,c) \;=\; \alpha_1\gamma_1 e^{-\gamma_1 c}\lambda_1(t) + \alpha_2\gamma_2 e^{-\gamma_2 t}\lambda_2(c) \tag{8.8}$$

where $\lambda(t,c)$ is the joint failure intensity function, $\lambda_1(t)$ is the failure intensity function with respect to time, while $\lambda_2(c)$ is the failure intensity function with respect to coverage. α_1, γ_1, α_2 and γ_2 are all parameters with the constraint of $\alpha_1 + \alpha_2 = 1$.

Since $\lambda_1(t)$ is the failure intensity function with respect to time, any existing distributions in well-known reliability models can be used, e.g., NHPP, Weibull model, S-shaped model and logarithmic Poisson models. Similarly, we can use any form such as the hyperexponential and Beta models proposed before for the failure intensity function with respect to coverage $\lambda_2(c)$.

To illustrate the detailed format of (8.8), if we use NHPP models for both time and coverage, we will get this joint failure intensity function:

$$\lambda(t,c) \;=\; \alpha_1\gamma_1 e^{-\gamma_1 c} N_1\beta_1 e^{-\beta_1 t} + \alpha_2\gamma_2 e^{-\gamma_2 t} N_2\beta_2 e^{-\beta_2 c} \tag{8.9}$$

From the integral of the failure intensity function in (8.9), we can get the expected cumulated number of failures when execution time is t, and cumulated coverage achieved is c:

$$\begin{aligned} F(t,c) \;=\;& \alpha_1(1-e^{-\gamma_1 c})N_1(1-e^{-\beta_1 t}) + \\ & \alpha_2(1-e^{-\gamma_2 t})N_2(1-e^{-\beta_2 c}) \end{aligned} \tag{8.10}$$

where $\alpha_1 + \alpha_2 = 1$.

On the other hand, if we use the Beta model for coverage, the joint failure intensity function will be :

$$\lambda(t,c) \;=\; \alpha_1\gamma_1 e^{-\gamma_1 c} N_1\beta_1 e^{-\beta_1 t} + \alpha_2\gamma_2 e^{-\gamma_2 t} N_2\beta_2 (1-c)^{\beta_2 - 1} \tag{8.11}$$

The expected cumulated number of failures is:

$$\begin{aligned} F(t,c) &= \alpha_1(1-e^{-\gamma_1 c})N_1(1-e^{-\beta_1 t}) + \\ &\quad \alpha_2(1-e^{-\gamma_2 t})N_2[1-(1-c)^{\beta_2}] \end{aligned} \tag{8.12}$$

We can prove the joint density function $f(t,c)$ derived from (8.8) by calculating its theoretical integral with respect to t and c and getting the result of 1.

$$\begin{aligned} \int_0^\infty \int_0^\infty f(t,c)dtdc &= \int_0^\infty \int_0^\infty \alpha_1\gamma_1 e^{-\gamma_1 c} f_1(t)dtdc + \\ &\quad \int_0^\infty \int_0^\infty \alpha_2\gamma_2 e^{-\gamma_2 t} f_2(c)dtdc \\ &= \int_0^\infty \alpha_1\gamma_1 e^{-\gamma_1 c}dc + \int_0^\infty \alpha_2\gamma_2 e^{-\gamma_2 t}dt \\ &= \alpha_1 + \alpha_2 \\ &= 1 \end{aligned}$$

8.3 Experimental Setup

To collect the time, coverage and failure data for reliability models with our own multi-version program versions and mutants, we use a super-program for testing and evaluation purpose. The concept of super-program is first proposed in [16] in order to apply the testing data of fault-tolerant software for reliability estimation.

In our experiment, the super-program is composed of all the 34 program versions which contain 426 mutants, being treated as 426 faults or failures. The coverage is measured against the super-program.

The testing procedure is described as follows:

1. Initialize the testing pool which contains the whole acceptance test set or operational test set;

2. Select a test case randomly from the testing pool;
3. Run the super-program according to three different testing strategies:
    (a) Run all the mutants at the same time, find those failed and delete them;
    (b) Select a program version randomly, run all the mutants within this version, record the mutants failed, and remove them from the super-program;
    (c) Select one mutant from all the mutants in all versions, remove it if it fails, otherwise go to 2;
4. Remove the current test case from the testing pool, go to 2;

From the three different selection strategies for program versions and mutants, we will get three different testing results. These testing data can be applied to this new reliability model for evaluation. Performance comparisons with other well-known reliability models, such as G-O, M-O, Musa Basic model, etc, can also be made based on the experimental data. Meanwhile, the estimation accuracy under three different selection strategies can be further investigated..

8.4 Experimental Evaluation

In this experimental evaluation, we adopt the first testing strategies out of the three stated above, i.e., run all the mutants at the same time and remove those failed. For the other two testing strategies, we will evaluate and compare their performance with the first one in our later empirical study.

To estimate the parameters in our reliability model, we have to deal with different reliability model with respect to time and coverage separately. To make our evaluation clearer, we adopt NHPP growth model for execution time, and exponential/Beta model for test coverage as the different failure rates. Other failure rates can also be adopted for further evaluations. Moreover, least-squares estimation (LSE) method is used for parameter estimation in our experiment.

For each of the evaluations, we use two different methods described as following:

Method A: first, the parameters in the NHPP model(N_1,β_1) are estimated separately, following by the estimation of coverage-related failure rate (N_2,β_2). After these four parameters are determined, the other three parameters)α_1, γ_1 and γ_2) are optimized with the joint failure rate;

Method B: all the seven parameters are optimized altogether to find the best estimation according to existing experimental data.

In the two estimation methods, method A seems more reasonable, since it sets the failure rates first and estimate the parameters of dependencies on the basis of determined failure rates. However, method B is also a necessary complement, as it tries to capture the dependency between time and coverage together with their different failure rates according to the current empirical data. In order to make the results more converge, in our evaluation, we always set the initial values in method B as the parameters estimated separately before.

8.4.1 Exponential coverage estimation

Supposing the defect coverage and test coverage follows the exponential relationship as shown in (8.1), we have the cdf expression as in (8.10). From LSE

Table 8.2: Estimated reliability parameters for exponential coverage model

Method	α_1	γ_1	N_1	β_1	γ_2	N_2	β_2	SSE
A	-1.3844	3.0819	380	0.87	1.5110	1475	0.39	93849
B	1.7713	0.824	380	11.716	0.121	1475	-0.082	14130
NHPP Model	-	-	380	0.87	-	-	-	279230

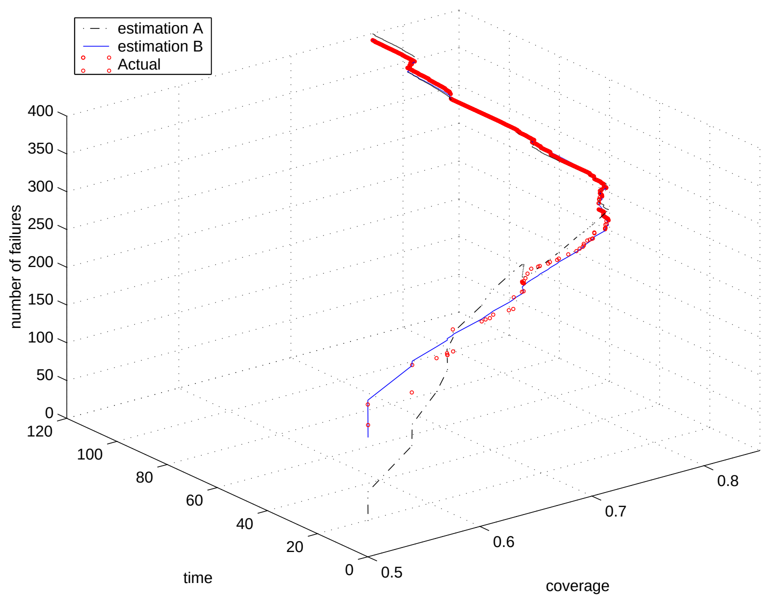

Figure 8.5: Reliability modeling with exponential failure rates

estimation for time and coverage separately, we get the parameters N_1, β_1 for time, and N_2 and β_2 for coverage. Using these known parameters and failure data in the experiment, we get the other three parameters using least-squares estimation, as listed in Table 8.2 and illustrated in Figure 8.5. The NHPP SRGM with respect to execution time is shown in Figure 8.6.

Table 8.2 shows that our estimation method A has an improvement compared with NHPP model, while estimation method B improves the estimation performance significantly. This result supports our previous observation: software failure behavior is not only related to testing time, but also test coverage

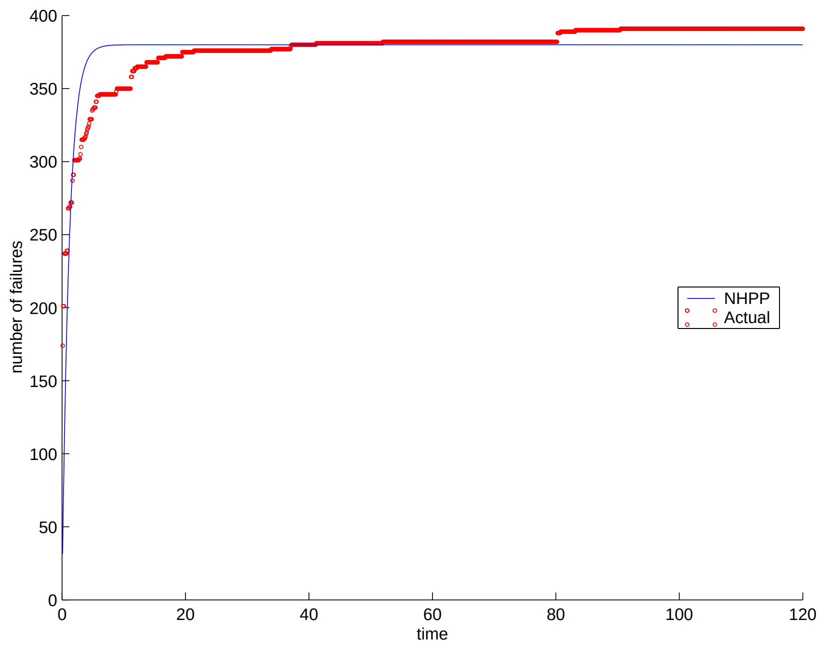

Figure 8.6: Reliability modeling with NHPP time relationship

or completeness. Although in our case, α_2 is a negative number since α_1 is larger than 1, both time and coverage contribute to the number of failures detected in the testing, and to the final reliability eventually.

8.4.2 Beta coverage estimation

Supposing the failure/coverage relationship follows the equation (8.7), we have the cdf expression shown in (8.12). Again, we use two estimation methods illustrated in previous section and compare their estimation performance in Table 8.3 and Figure 8.7.

From Figure 8.5 and Figure 8.7, it can be observed that the reliability estimation using our new reliability modeling is more accurate than that of NHPP model, especially for method B.

As shown in our evaluation, the advantage of our reliability model is: this

Table 8.3: Estimated reliability parameters for Beta coverage model

Method	α_1	γ_1	N_1	β_1	γ_2	N_2	β_2	SSE
A	0.0407	16.097	380	0.87	19.516	1101	0.303	36825
B	0.0565	20.182	380	0.098	21.138	1101	0.305	25712
NHPP Model	-	-	380	0.87	-	-	-	279230

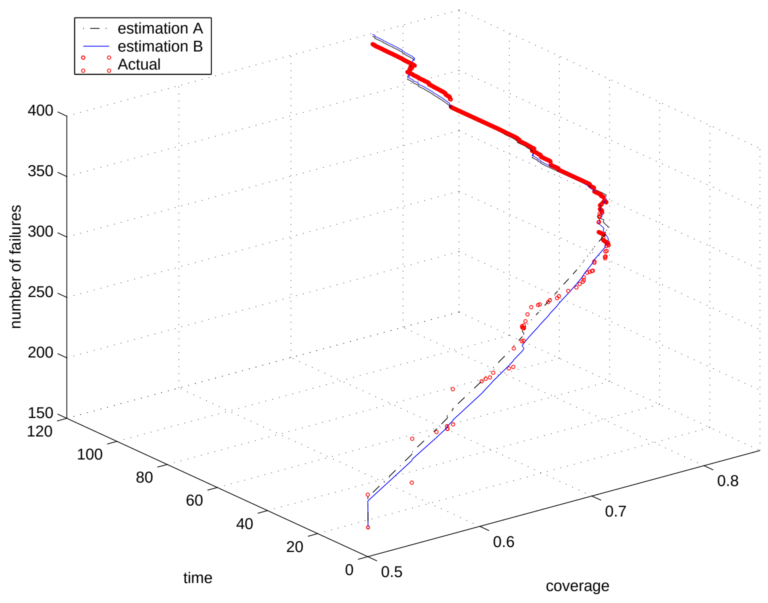

Figure 8.7: Reliability modeling with Beta coverage relationship

is the first time that test coverage information is introduced into software reliability modeling. It is based on the consideration that software failure behavior is not only related to execution time, but also to the test completeness. Our experimental results have shown that our reliability estimation is more accurate than that of NHPP software reliability model based on time domain.

Further comparisons and evaluations with other coverage models, such as Rayleigh distribution and exponential-logarithmic model, can be investigated. Moreover, as our future work, we will compare the prediction accuracy under three different version selection strategies described above.

8.5 Summary

In this chapter, we propose a new reliability model which integrates time and coverage measurements for reliability prediction. The key idea is that failure detection is not only related to the time that the software experiences under testing, but also how much the code fraction has been executed by the testing. This is the first time that execution time and test coverage are incorporated together to estimate the reliability achieved.

Our experimental results show that our reliability model gives as accurate estimation as NHPP model with an obvious improvement. Yet different coverage distributions do not affect the final estimation performance much with our experimental data.

As our future work, we will incorporate other coverage models into our reliability model and compare their performance. Furthermore, other SRGMs like M-O model and Musa Basic models can be adopted for performance comparison. Also, the other two testing strategies can be used to collect more failure data.

□ **End of chapter.**

Chapter 9

Conclusion and Future Work

In this thesis, we perform a comprehensive investigation and evaluation of fault-tolerant software including reliability modeling under fault correlation and intensive studies with software testing techniques. In order to provide quantitative assessment scheme for fault-tolerant software, we construct a comprehensive procedure in assessing fault-tolerant software for software reliability engineering, which contains four procedures: modeling, experimentation, evaluation and economics.

First, we propose a new software reliability model by introducing the information of test coverage achieved by test set. The test coverage measurement can be integrated into the time-domain reliability modeling, showing that the software reliability is not only related to testing time, but also to the code coverage which shows the test completeness.

Second, in order to evaluate such assumptions and other existing reliability models for fault-tolerant software, and to collect testing evidence for the evaluation, we conduct a large-scale, real-world, multiple-version software project. Software faults that are manifested in development and test are identified and studied. Coverage testing and mutation testing are executed on the multiple versions to collect testing data for the evaluation purpose.

Third, to evaluate the current reliability models for software fault tolerance under fault correlation, we apply our experimental data for predication effectiveness and performance comparison.

Furthermore, we perform comprehensive cross project comparisons on two large-scale projects: our project and NASA 4-University project, to investigate the "variants" and "invariants" features of fault-tolerant software. Although there are some common faults in the two projects because they are based on the same specification, the small number of coincident failures in both projects, nevertheless, provide a supportive evidence for N-version programming, while the observed reliability improvement implies some trends in the software development in the past twenty years.

Finally, we use our experimental data to investigate the effect of code coverage on fault detection and find that code coverage is a moderate indicator for the capability of fault detection on the whole test set. But 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, but weak in normal testing. Moreover, our study shows that code coverage can be used as a good filter to reduce the size of the effective test set, although it is more evident for exceptional test cases. Based on this observation, we are able to derive the software reliability growth model using the information of test coverage.

As our future work, the relationship between fault coverage and test coverage can be further formulated to improve the accuracy. The new reliability model can be further improved by considering other parameters which may influence the final reliability of fault-tolerant software.

In the meanwhile, although the current data are very comprehensive and valuable for other investigations and evaluations with respect to software reliability, software testing and software fault tolerance, more experiments can be setup and data can be collected based on current experimental environment.

Moreover, the final task in our research methodology - economics, i.e., the tradeoffs between software testing and fault tolerance, is still our future work. This will help us to complete our goal to construct a quantitative assessment scheme for fault-tolerant software.

□ **End of chapter.**

Appendix A

Publication List

Journal papers and book chapters

- **Xia Cai**, Michael R. Lyu and Kam-Fai Wong, *A Generic Environment for COTS Testing and Quality Prediction, Testing Commercial-off-the-shelf Components and Systems*, Sami Beydeda and Volker Gruhn (eds.), Springer-Verlag, Berlin, 2005, pp.315-347.

- Michael R. Lyu and **Xia Cai**, *Fault-tolerant Software*, To appear in Encyclopedia on Computer Science and Engineering, Benjamin Wah (ed.), Wiley. .

- **Xia Cai**, Michael R. Lyu, *An Experimental Evaluation of the Effect of Code Coverage on Fault Detection*, Submitted to IEEE Transactions on Software Engineering, June 2006.

- **Xia Cai**, Michael R. Lyu, Mladen A. Vouk, *Reliability Features for Design Diversity: Cross Project Evaluations and Comparisons*, in preparation.

- **Xia Cai**, Michael R. Lyu, *Predicting Software Reliability with Test Coverage*, in preparation.

Conference proceedings

- **Xia Cai**, Michael R. Lyu and Mladen A. Vouk, An Experimental Evaluation on Reliability Features of N-Version Programming, Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE2005), Chicago, Illinois, Nov. 8-11, 2005, pp. 161-170.

- **Xia Cai** and Michael R. Lyu, The Effect of Code Coverage on Fault Detection under Different Testing Profiles, ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST), St. Louis, Missouri, May 2005.

- **Xia Cai** and Michael R. Lyu, An Empirical Study on Reliability and Fault Correlation Models for Diverse Software Systems, Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE2004), Saint-Malo, France, Nov. 2004, pp.125-136.

- Michael R. Lyu, Zubin Huang, Sam K. S. Sze and **Xia Cai**, An Empirical Study on Testing and Fault Tolerance for Software Reliability Engineering, Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003), Denver, Colorado, Nov. 2003, pp.119-130.

  This paper received the ISSRE'2003 Best Paper Award.

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

[3] A. Avizienis. The methodology of n-version programming. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 23–46. Wiley, New York, 1995.

[4] 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.

[5] A. Avizienis, M. R. Lyu, and W. Schutz. In search of diversity: a six-language study of fault-tolerance flight control software. In *Proceedings of FTCS-18*, pages 15–22, Tokyo, Japan, 1988.

[6] B. Baudry, F. Fleurey, and Y. L. Traon. Improveing test suites for efficient fault localization. In *Proceedings 28th International Conference on Software Engineering (ICSE'2006)*, pages 82–90, Shanghai, China, May 2006.

[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*. 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] P. Boland, H. Singh, and B. Cukic. Comparing partition and random testing via Majorization and Schur functions. *IEEE Transactions on Software Engineering*, 29(1):88–94, January 2003.

[13] 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.

[14] X. Cai and M. R. Lyu. An empirical study on reliability and fault correlation models for diverse software systems. In *Proceedings 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'2004)*, Saint-Malo, France, November 2004.

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

[16] M. H. Chen, M. R. Lyu, and E. Wong. Incorporating code coverage in the reliability estimation for fault-tolerant software. In *Proceedings 16th IEEE Symposium on Reliable Distributed Systems*, pages 45–52, Durham, North Carolina, October 1997.

[17] 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.

[18] T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, 20:977–980, December 1994.

[19] 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.

[20] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong. Orthogonal defect classification – a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(19):943–956, November 1992.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10:438–444, July 1984.

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

[28] 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.

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

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

[31] P. G. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, August 1998.

[32] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification*, pages 154–164, October 1991.

[33] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19:202–213, March 1993.

[34] 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.

[35] M. R. Gary and D. S. Johnson. *Computers and Intractability.* Freeman, New York, 1979.

[36] S. Gokhale, M. R. Lyu, and K. S. Trivedi. Incorporating fault debugging activities into software reliability models: A simulation approach. *IEEE Transactions on Software Engineering*, 55(2):281–292, June 2006.

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

[38] 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.

[39] 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.

[40] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16:1402–1411, December 1990.

[41] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

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

[43] J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12:124–133, Januray 1986.

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

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

[46] W. E. Howden. Functional programming testing. *IEEE Transactions on Software Engineering*, 6, 1980.

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

[48] 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.

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

[50] B. Jeng and E. J. Weyuker. Some observations on partition testing. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pages 38–47, December 1989.

[51] P. C. Jorgensen. *Software Testing: A Craftsmans Approach (2nd edition).* CRC Press, 2002.

[52] 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.

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

[54] 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 18th International Symposium on Fault-Tolerant Computing*, pages 9–14, June 1988.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

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

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

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

[67] B. Littlewood, P. Popov, L. Strigini, and N. Shryane. Modelling the effects of combining diverse software and fault removal techniques. *IEEE Transactions on Software Engineering*, 26(12):1157–1167, 2000.

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

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

[70] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1996.

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

[72] M. R. Lyu, J. R. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing. In *Proceedings of ISSRE'93*, pages 25–34, Denver, November 1993.

[73] 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.

[74] 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.

[75] R. Maier, G. Bauer, G. Stoger, and S. Poledna. Time-triggered architecture: a consistent computing platform. *IEEE Micro*, 22(4):36–45, July/August 2002.

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

[77] Y. K. Malaiya, L. Naixin, J. Bieman, R. Karcich, and B. Skibbe. The relationship between test coverage and reliability. In *Proceedings of 5th International Symposium on Software Reliability Engineering*, pages 186 –195, November 1994.

[78] S. Morasca and S. Serra-Capizzano. On the analytical comparison of testing techniques. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 154–164, Massachusetts, USA, July 2004.

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

[80] 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.

[81] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, October 2001.

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

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

[84] 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.

[85] 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.

[86] 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.

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

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

[89] B. Randell. System architecture for software fault tolerance. *IEEE Transaction on Software Engineering*, 7(6):220–232, June 1975.

[90] 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.

[91] 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.

[92] RCS. *http://www.gnu.org/software/rcs/rcs.html/.* GNU project - Free Software Foundation (FSF), 2006.

[93] W. W. Royce. Managing the development of large software systems:concepts and techniques. In *Proceedings of IEEE WESTCON*, pages 1–9, Los Angeles, 1970.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] L. A. Tomek. Analyses using stochastic reward nets. In M. R. Lyu, editor, *Software Fault Tolerance*. Wiley, New York, 1995.

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

[100] 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.

[101] 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.

[102] M. A. Vouk. Using reliability models during teting with nonoperational profiles. In *Proceedings 2nd Bellcore/Purdue Workshop on Issues in Software Reliability Estimation*, pages 103–111, October 1992.

[103] 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 multiversion software development experiment. In *Proceedings of Digital Avionics Systems Conference*, pages 378–385, October 1990.

[104] E. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.

[105] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17:703–711, July 1991.

[106] 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.

[107] 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.

[108] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. *Journal of Systems and Software*, 1999.

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

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

[111] H. Zhong, L. Zhang, and H. Mei. An experimental comparison of four test suite reduction techniques. In *Proceedings 28th International Conference on Software Engineering (ICSE'2006)*, pages 636–640, shanghai, China, May 2006.