Nuclear Engineering Education Research
(NEER)
Grant Program

Development of a Method for Quantifying the Reliability of Nuclear Safety-
Related Software

Annual Report

by

Yi Zhang, Michael W. Golay

Department of Nuclear Engineering
Massachusetts Institute of Technology

October 1, 2003

Table of Contents

SUBJECT	PAGE
List of Figures_____	6
1. Introduction_____	1
2. Work Summary_____	1
3. Software Testing Method Literature_____	1
3.1. Software Testing Method Literature_____	1
3.2. A Mixed Testing Technique— Gray Box Method_____	9
3.2.1. Method Overview_____	10
3.2.2. More on Gray Box Testing Technique_____	10
3.2.3. Conclusion_____	13
3.3. Reliability Estimation Methodologies_____	13
3.3.1. Software Reliability Model Review_____	13
3.3.1.1. Black Box Reliability Models_____	14
3.3.1.2. White Box Reliability Models_____	15
3.3.2. Reliability Estimation Methods Developed in this study_____	15
3.3.2.1. Complete Nodal Coverage Reliability Estimation_____	15
3.3.2.2. Method General_____	15
3.3.2.3. The Initial Incentive_____	16
3.3.2.4. Methodology Details_____	16
3.3.2.4.1. Assumptions_____	16
3.3.2.4.2. The Formula_____	18
3.3.2.5. Methodology Conclusion_____	23
3.3.3. Refined Feasible Flowpath Coverage Reliability Estimation_____	24
3.3.3.1. Method General_____	24
3.3.3.2. Method Details_____	25
3.3.3.2.1. Assumptions_____	25
3.3.3.2.2. The General Formula_____	26
3.3.3.2.3. Estimate the Average Unreliability of the Tested Flowpaths_____	26
3.3.3.2.3.1. Two Types of Software Defects_____	27
3.3.3.2.3.2. Update Probability Distributions Associated with The Two Error Types_____	29
3.3.3.2.3.3. Estimate Reliability of a Tested Flowpath_____	35

3.3.3.2.3.4. Summary — Estimate Unreliability of tested flowpaths	40
3.3.3.2.4. Estimate the Average Unreliability of the Untested Flowpaths	41
3.3.3.2.4.1. Similarities And Differences between Tested and Untested Flowpaths	42
3.3.3.2.4.2. Unreliability of Untested Flowpaths	43
3.3.3.2.5. Estimate the Number of Possible Flowpaths, $p_{visit,T}$ and $p_{visit,U}$	49
3.3.3.2.5.1. Unfeasibility of Theoretical Solution	49
3.3.3.2.5.2. First Experimental Solution	52
3.3.3.2.5.3. Second Experimental Solution	56
4. Demonstration	59
4.1. Introduction	59
4.1.1. General	60
4.1.2. Working Path Flow	60
4.2. Sample Software — Signal Validation Algorithm (SVA)	61
4.2.1. SVA — the Algorithm	61
4.2.2. Original Design Document of SVA	62
4.2.3. Oracle	62
4.2.4. A Testing Document of SVA	62
4.3. 001 CASE Tool	63
4.3.1. DBTF	63
4.3.2. 001 System	63
4.4. Capture SVA Specification into 001 TMaps and FMaps	63
4.4.1. Formal Structures Building	64
4.4.2. Specification Incompleteness and Inconsistency Captured	64
4.4.3. Modular Testing	64
4.5. Integrated Testing on SVA	65
4.5.1. Testing Data Generation	66
4.5.1.1. Input Identification from Operational Profile	66
4.5.1.2. Random Sampling	66
4.5.2. Automatic Testing Process	71
4.5.2.1. Output Recording	72
4.5.2.2. Automatic Output Comparison	73
4.5.3. Regression Testing	73
4.5.4. Testing Results	73
4.6. Reliability Estimation	73

4.6.1. Required Information_____	73
4.6.2. Flowpath Recording in 001_____	74
4.6.2.1. The Nodes in 001_____	74
4.6.2.2. Feasible Paths — Cutting Extra Iterations of Loop Structures_____	75
4.6.3. Reliability Estimation Process Overview_____	75
4.6.4. Complete Nodal Coverage Approach_____	76
4.6.4.1. Node Visiting Frequency and Uncovered Nodes_____	76
4.6.4.2. Testing Results, from Nodes Point of View_____	77
4.6.4.3. Reliability Estimation from Nodal Coverage Approach_____	77
4.6.5. Refined Complete Feasible Path Testing Approach_____	80
4.6.5.1. Truncate Extra Iterations of Looping Structures_____	81
4.6.5.2. Distinct Feasible Flowpath Identification_____	82
4.6.5.3. Count Visiting Frequency of Flowpaths_____	83
4.6.5.4. 001 SVA Reliability Estimation Based on Feasible Flowpath Coverage_____	83
4.6.5.4.1. Estimate Unreliability of the Tested Flowpaths_____	83
4.6.5.4.1.1. Percentage of Type I Error in 001 SVA_____	84
4.6.5.4.1.2. Probability of encountering Type II Error in 001 SVA_____	85
4.6.5.4.1.3. Estimate Tested Flowpath Unreliability by Mean Value in 001 SVA_____	86
4.6.5.4.1.4. Estimate Tested Flowpath Unreliability by Gamma Distribution_____	89
4.6.5.4.2. Estimate the Number of Possible Flowpaths, $p_{visit,T}$ and $p_{visit,U}$ for 001 SVA_____	91
4.6.5.4.2.1. Unfeasibility of the First Experimental Solution_____	91
4.6.5.4.2.2. Estimate from the Second Experimental Solution_____	91
4.6.5.4.3. Estimate Unreliability of the Untested Flowpaths_____	93
4.6.5.4.4. Estimate Average Unreliability of the Whole Software_____	93
4.6.6. Results Summary_____	94
5. Appendix_____	95
5.1. Arithmetic Average Vs. Geometric Average_____	95
5.2. SVA Input Data Sets Description_____	96
5.3. Original Oracle SVA Interface_____	97
5.4. Oracle SVA Input Data Set Files_____	97
5.5. Input Data File Including Intermediate Parameter Values_____	98
5.6. TMaps Constructed for SVA 001 Input Data_____	99
5.7. Output Files from Oracle SVA Program (“oracle#.dat”)_____	100
5.8. SVA Errors Overview_____	101

5.9. Calculate Node Visiting Frequency_____	101
5.10. Untested Nodes on 001 SVA Program after 78,717 Tests_____	103
5.11. Cut Extra Loop Repetitions_____	106
5.12. Steps to Identify Unique Flowpaths_____	108
5.13. Count Visiting Frequency of Flowpaths_____	110
6. References_____	113

List of Figures

SUBJECT	PAGE
Figure 3-1 Nodal Coverage Marking Process1_____	12
Figure 3-2 Nodal Coverage Marking Process2_____	13
Figure 3-3 Venn diagram with events A and $E_1, E_2, \dots, E_n$ _____	30
Figure 3-4 Six Failure Observations due to One Error_____	43
Figure 3-5 Ratio of Observed Reliability over Intrinsic Reliability for Untested Flowpaths____	45
Figure 3-6. Calculate Number of Flowpaths Theoretically_____	50
Figure 3-7 Ants and Maze_____	52
Figure 3-8 Identify Possible Flowpaths_____	53
Figure 3-9 Estimate Total Number of Flowpaths, Experimental Approach1_____	54
Figure 3-10 Predicted Total Flowpath Number $\hat{N}$ Smaller than Tested Flowpath Number N_T _____	56
Figure 3-11 Ideal Number of Visits to Flowpaths According to Identification Order_____	57
Figure 3-12 Estimate Untested Flowpath Visiting Probability $p_{visit,U}$ _____	58
Figure 4-1 SVA Demonstration Working Path_____	61
Figure 4-2 Four arrangement settings for one SVA input data set_____	69
Figure 4-3 Cut Extra Iterations of a Looping Structure_____	75
Figure 4-4 Number of Visits to SVA Nodes after 198,321 Tests_____	77
Figure 4-5 SVA Unique Node Unreliability_____	79
Figure 4-6 Cut Iteration in Flowpaths_____	81
Figure 4-7 Number of Test Cases vs. Number of Flowpaths, SVA_____	82
Figure 4-8 Visiting Frequency of Unique 001 SVA Flowpaths_____	83
Figure 4-9 U.S. Average Software Defect Potentials_____	88
Figure 4-10 Tested Flowpath Unreliability in 001 SVA Program, Average Value Approach____	89
Figure 4-11 Tested Flowpath Unreliability in 001 SVA Program, Distribution Approach____	90
Figure 4-12 Flowpath vs. Test Case of 001 SVA Program_____	91
Figure 4-13 Number of Visits vs. Flowpath Index for 001 SVA_____	92
Figure 5.3-1 Oracle SVA Program Initial Screen_____	97
Figure 5.3-2 Oracle SVA Program Output Screen_____	97
Figure 5.6-1 TMaps Storing Input Data for 001 SVA Program_____	100
Figure 5.11.1-1 A Static Unexpanded 001 FMap Containing Looping Structure_____	106

Figure 5.11.1-2 A Flowpath with Three Repetitions of a Looping Structure_____	107
Figure 5.12-1 Identify Unique Flowpaths Step by Step_____	109

1. Introduction

The work of our project is intended to help introducing digital technologies into nuclear power into nuclear power plant safety related software applications. In our project we utilize a combination of modern software engineering methods: design process discipline and feedback, formal methods, automated computer aided software engineering tools, automatic code generation, and extensive feasible structure flow path testing to improve software quality.

The tactics include ensuring that the software structure is kept simple, permitting routine testing during design development, permitting extensive finished product testing in the input data space of most likely service and using test-based Bayesian updating to estimate the probability that a random software input will encounter an error upon execution. From the results obtained the software reliability can be both improved and its value estimated. Hopefully our success in the project's work can aid the transition of the nuclear enterprise into the modern information world.

In our work, we have been using the proprietary sample software, the digital Signal Validation Algorithm (SVA), provided by Westinghouse. Also our work is being done with their collaboration. The SVA software is used for selecting the plant instrumentation signal set which is to be used as the input the digital Plant Protection System (PPS). This is the system that automatically decides whether to trip the reactor. In our work, we are using OO1 computer assisted software engineering (CASE) tool of Hamilton Technologies Inc. This tool is capable of stating the syntactic structure of a program, reflecting its stated requirements, logical functions and data structure.

2. Work Summary

While improving the grey box testing approach, the work of the past year has been focused on reliability estimation method development. Two Bayesian updating based white box reliability estimation approaches were designed, implemented and demonstrated. Serving as an indispensable part of the grey box testing method, the nodal coverage based approach is also a handy and quick way of approximating software reliability. The flowpath coverage based approach can provide more precise numerical value, and more importantly, while following the estimation process, various fundamental features in terms of the reliability of the given software are able to be exposed. To make better estimation of the underlying flowpath reliability, two error types are introduced. A new experimental technique is designed to estimate how much of the program that has not been examined.

3. Grey Box Testing

3.1. Software Testing Method Literature

Testing is the process of executing a program on a set of test cases and comparing the actual results with the expected results. Its purpose is to reveal the existence of errors.

One of the main objectives of this project is to find a testing technique proper for safety-critical software used in Nuclear Power Plants. The resulting method should be as complete as possible due to the safety feature of the software in question while kept practical enough for real life use. In order to achieve this goal, the literature is extensively reviewed, and each method is rated for completeness and general usefulness. In the end, a new testing technique is introduced

incorporating features from the two major existing testing methods, giving the maximum degree of completeness while maintaining the testing workload reasonable.

A search of the literature of code testing reveals many different testing strategies. Some of these methods are specific to certain program types or programming languages and are ignored due to their lack of generality. In this study, the final chosen method should be useful in general and not in specific cases only. Many of the methods found fit this criterion. Although the methods are quite diverse, they can be broken down into two categories: black box testing and white box testing. Black box methods are based purely upon the specification document of the program in question. The code is treated as a closed box and is never examined. In white box testing, the code is examined, and tests are formulated based on some aspect of the code itself.

Black box testing methods, also called functional testing methods, derive test cases from the specifications without regard for the internal structure of the module being tested. The tester is not concerned with the mechanism, which generates an output, only that the output is correct for the given input set. Simple programs are often tested with black box methods.

White box testing methods generate test sets by examining the inner workings of the program in question. White box testing is also known as structural testing due to the basis of the method upon the code structure. While black box testing is generally the best place to start when attempting to test a program thoroughly, it is rarely sufficient. Without looking at the internal structure of a program, it is impossible to know which test cases are likely to give new information. It is therefore impossible to tell how much coverage we get from a set of black box test data. It is necessary to also do a white box testing, in which the code of the program being tested is taken into account. Many types of structural testing methods exist. There are three basic structural testing techniques. They are statement, branch and path testing methods. They form the base of understanding most other forms of structural testing methods. These testing techniques are said to result in certain coverage of the code.

The goal of this literature review is to locate a close to complete method, taking into account its feasibility as well.

The two major groups of testing methods are black box testing and white box testing. Each method has its advantages and disadvantages. Among black box methods, the exhaustive input testing is known as complete. But implementation of exhaustive input testing results in an unreasonable amount of test cases. Among structural testing group, the complete path testing is declared to be complete, though it is not truly complete as we discussed in the previous section. But still the literature regards complete path testing as being impossible for all but the simplest loop free programs. Because of the unfeasibility of these methods, we conclude that there is not an existing practical method for testing all except very small simple programs.

Faced with the impossibility of exhaustive testing, the goal of this project becomes to find a reasonably small set of tests that will allow us to approximate the information we would have obtained through complete testing. We want to take advantage of the black box testing methods to check whether the correct functionality specified by the user has been achieved and also analyze the structure of the code itself to imply how much new information is given by certain test data. Reliability estimation techniques closely related to the integrated methods we choose are developed in order to approximate how complete we are, in terms of error detection, at certain testing stage, so that we can decide when we can terminate our testing process. In the following section, a framework of the testing method we have developed and demonstrated is discussed. It is a technique standing between white box and black box. Merits from both techniques are incorporated and their impracticalities circumvented.

3.2. A Mixed Testing Technique—Gray Box Method

Our own testing technique, gray box method, is presented in this section. It stands at point between black box testing and white box testing. We relate the two existing methods by

their nature connection, resulting in an approach, practical and satisfactory to our high reliability requirements.

3.2.1. Method Overview

The testing technique literature review tells us that there is no method on hand that fulfills the requirement for this study. We are intending to design a testing technique combining the advantages of current available methods. We use the black box method to generate input data since it can simulate the actual usage of the software in question given the specification document is always available for testing purpose. If some input data probability distribution could be obtained, Monte Carlo sampling would be a simple way to generate a large amount of black box input data and is an automated process by nature. The flowpath coverage testing technique is inversed so that it can be used in combination with the black box input data sampling process and its working speed increased dramatically. We look at the program itself as the software control flowgraph and assume it is always obtainable. The complete flowpath triggered by each test case is recorded as a sequence of nodes. The first goal of testing is to achieve a complete nodal or branch coverage of the program. The rough reliability estimation of the software could be estimated at this point. After this is achieved, we continue to test the program more thoroughly and through the later discussed reliability estimation technique to estimate the confidence we have for this software based on how much of the software has been tested in terms of flowpaths and the confidence level we have for each of them. Because the whole testing methodology is developed in parallel with an experiment and is shown to be suitable for the type of software in this study, described in the demonstration session later, we have no doubt about its feasibility. In the following More on Gray Box Testing Technique session, the testing process as a whole is presented more extensively. More coverage on all the particular issues concerning testing is given from section 3.3.2 to the end of section 3.3.

3.2.2. More on Gray Box Testing Technique

Every testing process starts from identifying input data sets. During the previous work of this project, the process of searching input data sets by examining the inner structure of the code has been studied carefully (Litt92). The formal process includes five steps. It takes in a program, and through thorough analysis to systematically produce test cases, which achieve structured path testing coverage involving no infeasible test cases. In brief, the five steps (Lung96) are program control flowgraph construction, dataflow analysis and transitional variable identification, further expansion of the flowgraph with respect to the transitional variables, McCabe's metric calculation to determine number of test cases needed and finally determination of values of the input variables that cause each test path to be executed. At a glance, this is a very straightforward method. But when confronting the issue of automation, the complexity, both time and space, associated with its application is still very high. It does not guarantee finding a feasible solution even if such a solution exists, especially when nonlinear constraints on the input data are present, let alone the case that some paths can never be accessed due to the data relationships.

In order to solve the above dilemma, we decide to adopt the trial and check method. It is well understood that how to generate input data without examining the source code both manually and in an automatic manner. We argue that due to the safety-critical feature of the software under our study, we should carefully design a set of input data from the software specification. Possibly, manual input identification is applied during this stage. But this is very important and is the first step for us to earn enough confidence over the software under those well-known operating scenarios. Secondly, with some operating profile of the input data present, Monte Carlo method could be used to computerize the process of generating input data. Monte Carlo Method is a numerical method that provides approximate solutions to a variety of mathematical

problems by performing statistical sampling experiments on random variables using a computer. The input variables to the Monte Carlo method are random variables with certain probability density distributions. In each single trial of Monte Carlo method, the values of input variables are generated randomly with respect to the probability density distributions, the system responses to the generated input variables are computed and the outcomes recorded. There are several benefits of this random sampling process. First the obstacle to automate the process is gone inherently. Second, without studying the internal structure of the code, missing paths or logics are possibly detected. Last, the real time situation is reproduced as long as the probability distribution underneath is up to a good precision. Certainly, there is one last too-easy-to-get method for the input data production, that is, to get them directly from the user. Unlike other software input data, without the source code, no input could be produced. Many process control or data processing programs used in nuclear power plant are taking input data from sensor readings or other long time existing well-known plant indicators. If large amount of those real data could be obtained before the software is actually applied, they definitely serve as the best testing input data.

Until now, we have described the trial portion of our trial and check approach. If we were to stop here, our approach would be a pure black box method. Now comes the check portion. As the target program is run with the generated input variables and the path executed is identified and recorded. Our first step in the check process is to complete branch coverage. The actual implementation is very simple. We take the program as the flowgraph. As each execution of the target program leads to a sequence of nodes — the executed path, for each tested path, all the nodes on the flowgraph is marked on the flowgraph. Each node is associated with a counter. Before any test is done, all counters are zero. Every time a node is found within a flowpath, the counter of that node is increased by one. If no visit has been made to a node, the value of its attaching counter is zero. Otherwise, the counter stores the number of visits that have been made to its corresponding node. After a certain amount of tests are done on the target program, the marking process is carried out for all the paths of interest. A list of all the nodes on the software in question, with the number of visits (the value of its counter) is output into a file. If no modification has been made through out the whole process, marking can be started from the first recorded flowpath after the last marking process. The latest node list file is combined with the previous node list file. Since the group of nodes has been changed, this job can be easily accomplished by adding counter values of the same node from both files together. If some modification has been made to correct some detected error, all the previous used input data should be applied to the target program again to make sure no new error has been introduced. As a result, the marking process has to be done for all the tested paths because the set of nodes may have been changed. Refer to figure 3-2 and figure 3-3 for the visualized description about the marking process. The marking job is done after certain amount of tests. When at a point, we find from the node list file that there are only a few nodes that have not been checked, the process to manually identify a particular set of inputs which will lead to the flowpaths that bypass those unvisited nodes should be performed. It is very possibly that some surplus nodes that are impossible to be visited should be found during this process. The tester then could decide whether those nodes should be deleted from the program. As shown in our demonstration result, that higher error rate would show up from the last few hard-to-visit nodes. The reason is that usually these nodes are dealing with rare event, which the programmer tends to not think very carefully about. After all the nodes have been tested at least once, a rough approximation of the software reliability could be estimated using the data obtained, the visiting frequency during testing for each node and number of errors found on each node, etc. This will be discussed later in the reliability estimation session. While dealing with a program belonging to the testable group (refer to section 3.1 for software categories according to their testability), as the one used in our demonstration work, it is very often than not that some extra, useless nodes can be found when trying to identify input data for every branch, or node, in the software. This is also a very

beneficial byproduct because those unreachable nodes can do nothing good except adding some confusing to the program.

Figure 3-1 Nodal Coverage Marking Process1

Flowgraph is marked after 1-2-3-5-6-8-10-12 is visited

Node : Number of Visits : Total Paths

1 : 2 : 2
1-2 : 1 : 2
1-2-3 : 2 : 2
1-2-3-4 : 1 : 2
1-2-3-5 : 1 : 2
1-2-3-5-6 : 1 : 2
1-2-3-5-7 : 0 : 2
1-2-3-4-8 : 2 : 2
1-2-3-4-8-9 : 1 : 2
1-2-3-4-8-10 : 1 : 2
1-2-3-4-8-11 : 0 : 2
1-2-3-4-8-9-12 : 2 : 2

Figure 3-2 Nodal Coverage Marking Process2

Flowgraph is marked after 1-3-4-8-9-12 is visited and node list shown is stored in a file

After the complete nodal coverage has been achieved on the program under examination, the same testing process can be continued depending on the reliability requirement. It should be notified that reliability estimation could be made at any testing stage whether or not a complete nodal coverage has been obtained. During the testing process, the software reliability based on path reliability is estimated at certain stages depending on various factors, e.g. the number of new errors detected, the speed of identifying new flowpaths, the estimation of the total number of executable flowpaths, etc. The detailed method of path based reliability estimation is covered in later section. But there is one piece of very important information better to be mentioned here is that when estimate the reliability, we do not consider every flowpath has the same importance. Since we assume the program is tested according to the actual operating profile, or a very close approximate, we put more weights on the more frequently visited flowpaths while testing. This is another piece of evidence showing that we are not blindly after complete path coverage. It is treated as one of several indicators showing how complete the testing process has been. We can get the message about how many more test cases (sets of input data) we need in order to test how much fraction of the total flowpaths. If some decent reliability level has been achieved and a lot of new input data are needed to cover a few untested flowpaths that are rarely visited in reality, we may very much decide to stop the testing process since the big effort having to be made is not worthwhile.

3.2.3. Conclusion

In this section, we have discussed the framework of the gray box testing technique, used in this project. In the next sections, the task of testing safety-critical software is addressed from several perspectives, determining the operational profile, generating test cases, modifications in the program, checking the correctness of the output, speeding up the testing process, reassessment after modification, incorporating verification, making proper continuity assumptions, etc. Clearly, some of the obstacles in testing such software can be overcome via advances in technology. Some of them that are more fundamental in nature remain unsolved.

3.3. Reliability Estimation Methodologies

3.3.1. Software Reliability Model Review

Software Reliability is one of the most important parameters of software quality. In Encyclopedia of Computer Science, it is defined as “The probability that a software fault that causes deviation from the required output by more than a specified tolerance, in a specified environment, does not occur during a specified exposure period”. In reality, people appear to not be able to endure any error at all and more than too often simply set their tolerance to error-free, especially under safety-critical circumstances. The software reliability is then simply quoted as “The probability of failure-free operation in a specified environment for a specified period of time” (Lyu96). A software failure occurs when the behavior of the software departs from its specifications, and it is the result of a software fault, a design defect, being activated by certain input to the code during its execution. During our study, we consider a more simplified situation than mentioned in the above two reliability definitions. Instead of considering a continuous time software system, we view our target program as a discreet demand-based system. Opposed to approximating the error-free probability during some time period, we focus on the probability that no error will occur on the next execution of the program. It is apparent that the demand-based

reliability estimation can be converted to the widely used continuous time-based reliability by multiplying the average execution time per demand factor.

Software reliability analysis is performed at various stages during the process of engineering software, for a system, as an attempt to evaluate if the software reliability requirements have been (or might be) met. The analysis results not only provide feedback to the designers but also become a measure of software quality. Statistical inference techniques and reliability models are applied to failure data obtained from testing or during operation to measure software reliability. The same as testing techniques, software reliability models are classified as being black box models and white box models. The difference between the two is simply that the white box models consider the structure of the software in estimating reliability, while the black box models do not. In the following two sections, a summary of fundamental black box and white box software reliability models are reviewed. Both from their names and their essential characteristic difference, they seem to have exactly the same relationship as black box testing and white box testing methods. But there is a significant change, that is, these two types of reliability analysis methods are targeting at two levels of the software systems (although the black box method can be applied to any software from its bottom level, the white box method has only been applied to highly modularized software, starting from its well-known, easy to analyze component level) while the two types of testing techniques are trying to solve the same problem. Uniformly assuming the component reliabilities are available, the white box reliability models are always working on top of the black box models.

3.3.1.1. Black Box Reliability Models

The *black box models* include failure rate models, error seeding models, curve fitting models, Bayesian models, etc.

The *failure rate models* emphasize on studying the per fault failure rate during the failure intervals. The models in this group try to capture how the failure rate changes as more faults are detected and corrected. Estimations on total number of errors in the software and failure rate are made based on the data collected during testing process. Predictions about the performance of the software in the future are made according to the underlying failure rate function and the estimated parameters.

The class of *error seeding models* estimates the total number of errors in the given software by introducing bugs on purpose. Tests are performed on the software, which contains both inherent errors and induced errors. As testing going on, faults from both groups will be discovered. The estimation of the total number of inherent faults is made from the number of seeded errors and the ratio of the two types of errors encountered.

The *curve fitting models* use statistical regression method to study the relationship between software complexity and the number of errors in the program, the number of changes and the failure rates, etc. The form of relationship function is proposed and the corresponding coefficients in the function are estimated by regression methods or time series analysis. This category can be divided into several groups based on the software reliability parameter that is after, e.g. total number of errors in the software, the complexity measure of the software or the failure rate of the software, etc.

The last group of black box model is *Bayesian model*. Both of the two reliability models used in this project are fundamentally Bayesian model. This group of model views reliability growth and prediction in a Bayesian framework rather than the traditional ones considered in the previous sections. The power of Bayesian updating strategy is that it can include various forms of information, both subjective expert judgments and objective experimental results, into one model. The previous models allow changes in the reliability only when an error occurs. In a Bayesian model, the reliability can be updated even after some error-free tests, reflecting the

growing confidence in the software by the user. The reliability is therefore a reflection of both the number of faults that have been detected and the amount of failure-free operation.

3.3.1.2. White Box Reliability Models

The white box software reliability models consider the internal structure of the software in the reliability estimation as opposed to black box models which only model the interactions of software with the system within which it operates. The contention is that black box models are inadequate to be applied to software systems in the context of component-based software, increasing reuse of components and complex interactions between these components in a large software system. Furthermore, proponents of white box models advocate that reliability models that consider component reliabilities, in the computation of overall software reliability, would give more realistic estimates.

The motivation to develop white box or architecture-based models includes development of techniques to analyze performability of software built from reused and Commercial-Off-The-Shelf (COTS) components, performing sensitivity analysis, i.e. studying the variation of application reliability with variation in component and interface reliability, and for the identification of critical components and interfaces. Within this group, there are two major model types: state based model and path based model.

In the *state based model* class, the control flowgraph is used to represent software architecture. Software reliability is estimated analytically. These models assume a control flowgraph has a single entry and a single exit node representing components at which execution begins and is terminated, respectively. The transfer of control among components is described by a transition probability matrix. There are two absorbing states, correct (C) and failure (F). The reliability of the program is the probability of reaching the absorbing state C.

Similar to state based models, *path based models* examine the software architecture explicitly and assume independent component failure. But instead of combining software architecture and software behavior analytically as in state based models, experimental methods are utilized. Different execution paths that can be taken are considered in path based models while components or nodes are examined in state based models.

3.3.2. Reliability Estimation Methods Developed in this study

3.3.2.1. Complete Nodal Coverage Reliability Estimation

3.3.2.2. Method General

Due to the safety-criticality of the software we are working on, the goal of reliability estimation is to provide a more precise software reliability measurement technique. It should be able to take as much advantage as the available information can provide. Also, it should be able to produce constructive feedbacks to the development process of the software. These objectives apparently lead us to white box approaches, which consider the internal structure of the software. As a natural extension of the first testing technique we discussed before, the first software reliability estimation method developed in this work is nodal coverage reliability estimation method. Instead of cutting the whole program into smaller modules, we take the natural elementary components of the software under consideration, the nodes, to estimate the software reliability. First a complete nodal coverage testing is achieved. Testing result for each node as while as its visiting frequency during testing is recorded. Unreliability of each node is estimated either after a reliability estimation terminating point or following modification and retesting of a node due to error detection. The unreliability of the software is calculated as visiting frequency weighed average of all nodes multiplied the average number of nodes visited per execution.

In the subsequent portions of this section, I will present the initial motivations, the assumptions and method itself in more details.

3.3.2.3. The Initial Incentive

The nodal coverage based reliability estimation is a by-product on our way toward a path coverage based reliability estimation methodology. It is relatively less demanding to achieve so that it can be applied to more complicated software than the programs we are studying in this project. We notice from our demonstration work that though the nodal approach is not as accurate, or using as much information, it still gives us a reasonable approximation.

There used to be a draft design on testing and reliability estimation process before the work described in this report was done. While we tried to implement the original design onto the real software sample, lots of problems concerning feasibility and automation showed up very quickly. The methodology presented in this report is way lot different compared to the original design due to this reason. Thus, it is the demonstration work that has been directing us towards the final approaches that are described in this thesis. This complete nodal coverage reliability estimation technique is one of those ideas that brought about this way.

As nodal coverage testing or branch coverage testing to path coverage testing, nodal coverage reliability estimation is a simplified version of path coverage reliability estimation approach. At the starting point of this project, due to the safety-critical feature of the target software used in nuclear power plants, we chose the goal of testing the software as complete as possible, that is, maintaining the feasibility at the same time. This goal leads us working toward a complete path coverage testing approach, which implies the reliability estimation technique is based on the paths also. While we are working on programs that are relatively simple and testable (section3.1), we used to believe that we could cover most of the paths on the program under question. But those thoughts turned out to be too idealistic when we encountered the situation of getting new paths steadily. New paths are the paths we have not met before during the testing process. We do not identify any paths before testing. The way we get to know, or identify a path is through testing. Every time a test is done, its passing flowpath is recorded. If we cannot find the same path in the group of paths we have tested before, it is announced a new path. At this stage, we had to answer a fundamental question, that is, how many paths in total do we have in the target program. We have to say we do not know. There are lots of existing paths that may not be reached by real input data at all.

We decide to make our tasks simpler at first by trying to test all of the nodes at least once. This checking process has been presented in 3.3.4.8. After we manage to complete a branch coverage testing, it seems a good time for us to look at reliability of the software from the branch or node point of view.

In this reliability estimation approach, unreliability is estimated for every node. The estimation is based on how many tests have been applied to a node and how many errors have been detected on that node. The overall software unreliability is approximated by the weighted summation of unreliability of every node in the software. Since we assume we are using input data with the same statistical properties as those that would be experienced in real operational use, the visiting frequency of every node during the testing process is used as the weight of that node while calculating the overall software unreliability. The reliability of the software is unit minus software unreliability. We are using unreliability instead of reliability during the calculation only for convenience. In the literature of safety-critical software, unreliabilities are very small such that less significant numbers are needed to describe the same level of software quality by using unreliability than reliability.

3.3.2.4. Methodology Details

3.3.2.4.1. Assumptions

In this section, the assumptions in the complete nodal coverage testing based reliability estimation method are listed and explained. They are as follows:

- 1) Testing is representative of actual use.
- 2) Faults are of the same severity.
- 3) Detected faults are fixed with certainty immediately.
- 4) Visits to the same node are statistically independent Bernoulli trials.

Testing is representative of actual use. This assumption says the testing is performed in a manner that is similar to intended usage. We want to estimate the software reliability by taking account of the importance of every node. What is the importance of a node? There are two values behind the concept of importance. The first one is how often a node is visited, or used, during operation. The more frequently a node is executed, the more important that node is. Since in this nodal coverage reliability estimation approach, we assume any error in a node is not path dependent, that is, if there is an error in a node, all the paths that bypass that node will fail. Then the first importance factor of a node is very simple. The more time it is visited, the more important a node is. Now if we try to do a test as close to the true situation as possible, the first importance factor for every node can be approximated by the visiting frequency of each node during testing process. This assumption ensures that the estimates that are derived using data collected in the testing environment are applicable to the environment in which the reliability projections are to be made.

Faults are of the same severity. There is another importance factor attached to each node. It is about how badly the result would be should a node fail, that is, the severity of a failure. This part of the importance of a node, unlike the visiting frequency of the node, which is an intrinsic feature of the code itself, it is related to the more fundamental phase of the software. We have to analyze the user requirement, the software specification and the code simultaneously and relate each of the actual functions to a node, or a group of nodes. What we know directly is how hazardous it is if some function fails. Apparently, this information is very software specific. Every program has to be analyzed manually to make some importance assignment. By taking into account the feasibility requirement of this study, we are not attacking this problem due to its hard-to-be automated nature. Also, since our software under question is relatively small and simply structured, there are not many functions involved, which leads to a very small number of different importance levels in terms of nodal failure consequence. Due to the above reasons, we assume every node has the same failure consequence severity.

Detected faults are fixed with certainty immediately. This is an assumption taken by all the software reliability growth models. In this project, rather than an assumption, it is a requirement to the development process of safety-critical software. As discussed in the testing section previously, the software used in nuclear industry has very high safety criterion that demands all detected faults removed. We ensure the realization of this requirement by full retesting. If an error is encountered, immediate corrections should be made and all the previously used input data should be used to test the modified software.

Visits to the same node are statistically independent Bernoulli trials. This assumption is about the testing process. It is certainly true that the probability of encountering an error on a less reliable node is much higher during every visit to that node. What the assumption means is that every test, or reliability measurement to a node is independent of all the other tests or reliability measurements to that node. The result of one measurement is not going to affect the result of any future measurement. We update reliability of each node by applying each reliability measurement result of that node independently to the updating model.

In summary, almost all the other reliability estimation models make the first three assumptions applied to this nodal coverage reliability estimation approach. The last two

assumptions are very nodal related and are specific to this reliability estimation technique. Both of them have very solid foundation for their existence.

Rather than inherit all the popular assumptions adopted by most of the software reliability models, this method has relaxed two standard assumptions made by most of the software reliability models.

The number of potential faults is fixed and finite. This assumption is not needed in our modal coverage based reliability estimation technique. In this approach, we predict the chance of encountering an error during the next execution on the software under study only based on the facts we have observed during the testing process. The feature that our reliability estimation model does not need an upper limit for the number of total errors greatly thanks to the Bayesian updating method. Only facts about how many numbers a node has been tested with or without error are used to estimate the reliability of that node.

Another relaxation to one of the traditionally used assumptions in software reliability models is about the possibility to introduce new errors while correcting the detected ones. In this approach, new faults can be introduced as a result of fixing existing faults. Opposed to many reliability estimation methods, we do take into consideration the possibility of introducing new errors due to the modification action. That is not to say we expect it to happen. In fact, we do our best to decrease of the possibility of bringing new errors by checking all the testing data used before. But it is not totally safe to say there are no new errors brought about by the modification to the untested part of the software. We still believe that the software reliability grows due to the modification. Instead of expressing the reliability growth in terms of total number of faults in the software reduced by one, we express it in a probability manner. If an error is detected during k^{th} visit to that node and modified immediately, we put the new information, that is, we have tested the node k times without an error, rather than $(k-1)$ times without an error, into the Bayesian updating model to update reliability of that node. By doing so, we do not exclude the possibility that we may encounter an error on this node during $(k+1)^{\text{th}}$ visit to this node as a result of some other input data due to the modification to this node. We do not exclude the possibility of detecting an error on another node in the future due to modification of this node. This different view we take on the error-modification consequence from most of the other known software reliability growth models makes this reliability estimation method based more on the facts rather than assumptions.

3.3.2.4.2. The Formula

This nodal coverage has a pre-requirement, that is, all the nodes are tested. This means complete branch coverage should have been achieved and every direction of each decision within the program is tested at least once. Upon completing that testing process, unreliability is estimated for each of the nodes in the program based on Bayesian theorem. As testing goes on, unreliability is updated using Bayesian updating method for every node. The overall unreliability of the software is approximated by the weighted summation of all the nodes within the software multiplied by the average number of nodes visited per test. In another word, the software reliability is a product of average unreliability over all nodes and average number of nodes reached per execution. The weight for each node, while calculating the average unreliability, is proportional to the visiting frequency of that node during testing. The formula for unreliability of the software is:

$$\theta_n = \frac{\sum_{i_n=1}^{N_n} p_{visit,i_n} \bar{\theta}_{i_n}}{\sum_{i_n=1}^{N_n} p_{visit,i_n}} \times \bar{x} N_n = \frac{\sum_{i_n=1}^{N_n} f_{visit,i_n} \bar{\theta}_{i_n}}{\sum_{i_n=1}^{N_n} f_{visit,i_n}} \times \bar{x} N_n \quad \text{Eq. 3-1}$$

θ_n	Software unreliability estimated by complete nodal coverage reliability estimation approach
N_n	Total number of nodes in the software
p_{visit,i_n}	Probability of node i_n being visited during real operation
f_{visit,i_n}	Visiting frequency of node i_n during testing process
$\bar{\theta}_{i_n}$	Mean unreliability of node i_n , estimated by Bayesian updating method
$\bar{x}$	Average fraction of nodes visited per test

Among the above parameters,

$$p_{visit,i_n} = f_{visit,i_n} = \frac{n_{tc,i_n}}{N_{tc}} \quad \text{Eq. 3-2}$$

n_{tc,i_n}	Number of test cases that bypass node i_n
N_{tc}	Total number of test cases

The only unknown within the above formula is the average unreliability for each node, $\bar{\theta}_{i_n}$, which is updated according to Bayesian updating method as tests proceed. Nodal unreliability is defined in the same manner as unreliability of the software. Unreliability of node i_n , θ_{i_n} , is the probability that node i_n fails if it is reached during the next execution.

We assume each visit to node i_n is a statistically independent Bernoulli trial. Thus, given θ_{i_n} , the number of failures on node i_n out of n visits, R_{i_n} , has a Bernoulli distribution:

$$P(R_{i_n} = r) = C_n^r \theta_{i_n}^r (1 - \theta_{i_n})^{n-r} \quad \text{Eq. 3-3}$$

Within Bayesian framework one represents one's prior knowledge about the parameter of interested, in this case θ_{i_n} , by the prior distribution. There are advantages in using a prior distribution from the conjugate family: it has the property that both prior and posterior distribution will be members of the same parametric family of distributions and thus represents a kind of homogeneity in the way in which one's belief change as one receives extra information. Here, the conjugate distribution chosen is the *Beta(a,b)* distribution:

$$f(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b-1}}{B(a, b)} \quad \text{Eq. 3-4}$$

where $B(a, b)$ is the Beta function and $a > 0, b > 0$ are chosen by the observer to represent his belief about θ_{i_n} prior to seeing any test results.

In some cases it might be possible to use information about the node and its development process to give numerical values for a and b . If no such information is available, “ignorance” uniform prior with $a = b = 1$ can be used:

$$f(\theta_{i_n}) = 1 \quad \text{Eq. 3-5}$$

If the node has been visited n times, and we have seen r failures on this node, the posterior distribution of θ_{i_n} is $Beta(a + r, b + n - r)$:

$$f(\theta_{i_n}) = \frac{\theta_{i_n}^{a+r-1} (1 - \theta_{i_n})^{b+n-r-1}}{B(a + r, b + n - r)} \quad \text{Eq. 3-6}$$

If $a = b = 1$, i.e. $f(\theta_{i_n}) = 1$, it is reduced to:

$$f(\theta_{i_n}) = \frac{\theta_{i_n}^r (1 - \theta_{i_n})^{n-r}}{B(1 + r, 1 + n - r)} \quad \text{Eq. 3-7}$$

There is a non-apparent assumption behind Bayesian updating method. The underlying parameter or parameter distribution is fixed while updated by Bayesian framework. Though this assumption is satisfied easily within most Bayesian updating applications, it is non-trivial in our project. In our safety-critical software case, it is required to remove all known faults. If a fault is detected and corrected, the underlying unreliability is changed, reduced in most cases. In this case, we cannot continue Bayesian updating method. Instead, we should re-construct the unreliability distribution from the very beginning, or, use all the information about the new unreliability and do Bayesian updating from the first prior distribution.

To be more specifically, we update or re-construct the unreliability distribution of node i_n as below:

- Prior Distribution of θ_{i_n} :

Before seeing any test results, express our belief about unreliability of node i_n as:

$$f_0(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b-1}}{B(a, b)}, \quad a > 0, b > 0 \quad \text{Eq. 3-8}$$

If there is no information about θ_{i_n} , $a = b = 1$

$$f_0(\theta_{i_n}) = 1 \quad \text{Eq. 3-9}$$

- Bayesian Updating θ_{i_n} after Δs_1 failure-free tests on node i_n :

$$f_1(\theta_{i_n}) = C_1 (1 - \theta_{i_n})^{\Delta s_1} f_0(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + \Delta s_1 - 1}}{B(a, b + \Delta s_1)} \quad \text{Eq. 3-10}$$

If $a = b = 1$

$$f_1(\theta_{i_n}) = \frac{(1 - \theta_{i_n})^{\Delta s_1}}{B(1, 1 + \Delta s_1)} \quad \text{Eq. 3-11}$$

- Re-construct distribution of θ_{i_n} after one failure on Δs_1^{th} visit to node i_n :

Now the underlying distribution of unreliability of node i_n has changed during to the correction made to the node. We denote the new unreliability of node i_n as θ'_{i_n} . Since we do not assume perfect correction in our safety-critical software case, the modified program is put on test again with all the previously used testing data. It is required that all known errors are removed. As the result of this requirement, after the modification and complete-retest process, we should observe Δs_1 failure-free visit to node i_n if no more new testing data are applied. This piece of information should be used as our only knowledge about distribution of θ'_{i_n} . This means we should update θ'_{i_n} from its very first prior distribution, which is the same as the first prior distribution of θ_{i_n} .

$$f_1(\theta'_{i_n}) = C_1 (1 - \theta'_{i_n})^{\Delta s_1} f_0(\theta'_{i_n}) = \frac{\theta'_{i_n}^{a-1} (1 - \theta'_{i_n})^{b + \Delta s_1 - 1}}{B(a, b + \Delta s_1)} \quad \text{Eq. 3-12}$$

And with $a = b = 1$, we have

$$f_1(\theta'_{i_n}) = \frac{(1 - \theta'_{i_n})^{\Delta s_1}}{B(1, 1 + \Delta s_1)} \quad \text{Eq. 3-13}$$

Though the formula is exactly the same as the result of Δs_1 failure-free test, the physics behind is totally different.

- Bayesian θ_{i_n} after Δs_2 further failure-free tests on node i_n :
Assume the distribution of θ_{i_n} becomes $f_1(\theta_{i_n})$ now. The underlying unreliability of node i_n has not changed and can be updated as follows:

$$f_2(\theta_{i_n}) = C_2 (1 - \theta_{i_n})^{\Delta s_2} f_0(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + \Delta s_1 + \Delta s_2 - 1}}{B(a, b + \Delta s_1 + \Delta s_2)} = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + s_2 - 1}}{B(a, b + s_2)} \quad \text{Eq. 3-14}$$

If $a = b = 1$

$$f_2(\theta_{i_n}) = \frac{(1 - \theta_{i_n})^{s_2}}{B(1, 1 + s_2)} \quad \text{Eq. 3-15}$$

- Re-estimate θ_{i_n} after Δs_2 further tests on node i_n with an error found on the last test:
Assume the distribution of θ_{i_n} becomes $f_1(\theta_{i_n})$ now.
If on test number $s_2 = \Delta s_1 + \Delta s_2$, an error is found. Correct this error and retest all the s_2 sets of input data until no error is found. Now we can say we have tested the program s_2 times without an error. The resulting new unreliability θ'_{i_n} has the same distribution as if the node had not been modified, but instead tested s_2 times without finding an error. The probability distribution of the unreliability of the modified node can be estimated as below. Since θ'_{i_n} is not the same as θ_{i_n} , we cannot use $f_1(\theta_{i_n})$ as our prior distribution for θ'_{i_n} . Instead, we update it from the very first prior distribution.

$$f_2(\theta'_{i_n}) = C_2 (1 - \theta'_{i_n})^{\Delta s_1 + \Delta s_2} f_0(\theta'_{i_n}) = \frac{\theta'_{i_n}^{a-1} (1 - \theta'_{i_n})^{b + \Delta s_1 + \Delta s_2 - 1}}{B(a, b + \Delta s_1 + \Delta s_2)} = \frac{\theta'_{i_n}^{a-1} (1 - \theta'_{i_n})^{b + s_2 - 1}}{B(a, b + s_2)} \quad \text{Eq. 3-16}$$

If $a = b = 1$

$$f_2(\theta'_{i_n}) = \frac{(1 - \theta'_{i_n})^{s_2}}{B(1, 1 + s_2)} \quad \text{Eq. 3-17}$$

- Average value of node unreliability, $\bar{\theta}_{i_n}$
Among all tests done on node i_n , after k stops, either due to an error or due to an intermediate estimation, the posterior distribution of its unreliability is:

$$f_k(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + \Delta s_1 + \Delta s_2 + \dots + \Delta s_k - 1}}{B(a, b + \Delta s_1 + \Delta s_2 + \dots + \Delta s_k)} = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + s_k - 1}}{B(a, b + s_k)} \quad \text{Eq. 3-18}$$

Among them,

Δs_k : The number of test cases processed between $(k-1)^{\text{st}}$ and k^{th} stops.
 $s_k = \Delta s_1 + \Delta s_2 + \dots + \Delta s_k$: Total number of test cases until k^{th} stop.

Again, if $a = b = 1$

$$f_k(\theta_{i_n}) = \frac{(1 - \theta_{i_n})^{s_k}}{B(1, 1 + s_k)} \quad \text{Eq. 3-19}$$

The average of θ_{i_n} is:

$$\bar{\theta}_{i_n} = \int_0^1 \theta_{i_n} f_k(\theta_{i_n}) d\theta_{i_n} = \frac{a}{a + b + s_k} \quad \text{Eq. 3-20}$$

If $a = b = 1$

$$\bar{\theta}_{i_n} = \int_0^1 \theta_{i_n} f_k(\theta_{i_n}) d\theta_{i_n} = \frac{1}{2 + s_k} \quad \text{Eq. 3-21}$$

3.3.2.5. Methodology Conclusion

Although, nodal coverage reliability estimation approach is not as precise as the later discussed flowpath based approach, it still has several advantages thanks to its simplicity.

First, it helps us to achieve a balanced test. By examining coverage status of each node, the un-tested nodes are picked out and input data created manually to cover them. As shown in our demonstration work, in doing this, errors on those scarcely visited nodes, which turn out to be useful, are detected much sooner than just using simple random input data. On the other hand, this process can lead us to those nodes that can never be reached and should be removed from the program. For the flowpath coverage approach, this function cannot be realized. While we can say if a node is in a program, it has to be visited by some input data, we cannot say the give the same statement to a flowpath. Some flowpaths have to stay in the program due to the usefulness of all the nodes on them, though no input data can go through those flowpaths. While it is not a very hard job to pick out the surplus components of the software by examining coverage status in terms of nodes, it is impossible in the flowpath coverage approach, let alone the unfeasibility of looking at the huge number of flowpaths within a program even for a testable case studied in this project.

Secondly, the nodal coverage estimation approach is still practical even the size of the software in question gets much bigger and its structure much more complicated.

Thirdly, the visiting frequency of each node gives us the importance information of that node. This is also a piece of information that cannot be obtained through analyzing the flowpaths of a program. With the critical parts of the program identified, more thorough testing can be applied to them. Still we can tell which flowpath is the most important one by its visiting frequency; we cannot test those important flowpaths individually. While nodes are exclusive components of the program, flowpaths are not. It is very hard to find those input data leading to a particular flowpath.

In summary, nodal coverage examination is irreplaceable in this study. It helps to achieve a balanced test before any estimation is done. Under the situation of relatively complicated software, nodal coverage estimation would be the substitute solution to the more precise path coverage estimation approach, discussed later in this thesis.

3.3.3. Refined Feasible Flowpath Coverage Reliability Estimation

As described in last section the first reliability estimation design in this project is based on flowpath coverage, nodal coverage due to the completeness criteria coming from the safe-critical nature of the software we are working on. When we faced the problem of unfeasibility, we relaxed our goal from flowpath-based to node-based method. After completing the nodal coverage based reliability estimation methodology demonstration, we start to wonder if we have taken advantage of all the information we have obtained through testing and if we can possibly get more constructive feedback to the testing process. Shall we stop testing after we have achieved complete nodal coverage / branch coverage on the program? The answer is definitely no. As we have recorded all the tested flowpaths, we should be able to do some analysis based on flowpaths. The stopping rule should be constructed in terms of flowpaths rather than nodes. The most important questions are the fraction of flowpaths that have been tested, how reliable the tested flowpaths are, how reliable the untested flowpaths are and how often they are going to be encountered, etc. All these questions are addressed in this refined feasible flowpath coverage reliability estimation methodology and exemplified in the sample software presented later in this thesis, hence its practicality proved.

3.3.3.1. Method General

In refined feasible flowpath coverage reliability estimation approach, the software is divided into two flowpath groups, the tested flowpaths and the untested flowpaths. The overall unreliability of the software is estimated by the weighed average of unreliability of the tested flowpath-group and the unreliability of the untested-flowpath group. The unreliability of the tested flowpath-group is approximated by a weighed average of the unreliability of all the flowpaths in this group and the unreliability of each tested flowpath is estimated from the testing result via Bayesian updating method. While we do not have any direct testing information of the untested flowpaths, we take advantage of the results obtained from the tested flowpaths assuming they are at the same unreliability level if no modification is made to either part because these two parts are designed, programmed by the same software developing team under the same environment. The weights, given to a flowpath or a group of flowpaths during the average calculation mentioned above, are always proportional to the probability that flowpath or that group of flowpaths will be visited during an execution of the program. For a tested flowpath, its visiting frequency during testing can be used as an approximation of its relative visiting probability within the tested group, or its weight within the tested group. For the untested flowpaths, there is no theoretical or experimental method we can use to measure how big it is, let alone which of them will be visited more than the others. Concerning the size of the group, diagram solution is used. The relative visiting probability of each member within the group is not

needed since an average unreliability value will be approximated from the information we get from the tested flowpaths.

In the following parts of this section, the assumptions and detailed formulas of this method are discussed.

3.3.3.2. Method Details

3.3.3.2.1. Assumptions

- 1) Testing is representative of actual use.
- 2) Faults are of the same severity.
- 3) Detected faults are fixed with certainty immediately.
- 4) Visits to the same flowpath are statistically independent Bernoulli trials.
- 5) If no modification has been made to the tested flowpaths, averagely, they are at the same reliability level as the untested flowpaths.
- 6) There are two possible types of errors on a flowpath, type I and type II. If an error is of type I, it is always detected during the first visit to the flowpath. If an error is of type II, it has equal opportunity to be detected during any visit to the flowpath.

The first assumption is the same as what we have used in the nodal coverage based reliability estimation approach. This assumption is made so that we can use the visiting frequency of each tested flowpath as its weight while calculating the average reliability of the tested flowpaths.

The second and third assumptions are exactly the same as in the nodal coverage approach, which have been discussed and are not repeated here.

The fourth assumption is almost the same as what have been practiced in the nodal approach except that rather than talking about different visits to a node, different visits to a flowpath are our targets. We take advantage of this assumption when updating reliability of a tested flowpath based on the testing results.

The fifth assumption says if we have only tested some flowpaths in the program but not correct any of the identified error, on average the tested flowpaths are at the same reliability level as the untested flowpaths. This is a fairly reasonable assumption based on the fact that there is no distinction between these two parts at all while they are developed. They are developed by the same group of people under the same developing environment. This assumption is made so that we can use the testing result obtained from the tested flowpaths to estimate the reliability of the untested flowpaths.

In this project, we have divided all software errors into two categories from the testing point of view. Type I error on a flowpath is always detected during the first time that flowpath is visited. Type II error on a flowpath has the same opportunity of being identified during any visit to that flowpath. Now there is a little difference about who an error belongs to from the nodal coverage context. The error belongs to a flowpath or a group of flowpaths. Since an error has to be within one node in order to belong to one or several flowpaths, it is important to notice that an error does not necessarily cause all the flowpaths that contain this node to be faulty. The reason is obvious. In our context, we have defined a node to be a group of statements. It is very possible that some statements within a node do not affect some of the flowpaths that go through that node. As a result, type I error is not necessarily to be identified during the first visit to the node that has that error.

Up to now, we have mentioned completeness in this thesis several times. Can we actually achieve a complete testing in terms of a perfect program? No, we are never one hundred percent sure that the program under testing is free of error. Even if a complete flowpath coverage has been performed on a piece of software, that is, every possible flowpath of it has been tested, it

is still possible that there is an undetected error on one of the tested flowpaths. Thus in this study, we do not assume a flowpath is free of error whether it has been tested one time or one hundred times. It is definitely true that the flowpath that has been tested one hundred times is much more stable than the flowpath that has been tested only once. If comparing our perspective on the possibility of error existing than the traditional approaches, ours is a more conservative one. But it is also true that this view is closer to the reality. Also, as can be seen in later section, the probability of error existing in a flowpath drops down very quickly as the number of testing on this flowpath increases.

3.3.3.2.2. The General Formula

The overall unreliability θ_p is calculated as the average of the average unreliability of the tested flowpaths $\bar{\theta}_{p,T}$ and average unreliability of the untested flowpaths $\bar{\theta}_{p,U}$ as:

$$\theta_p = p_{visit,T} \bar{\theta}_{p,T} + p_{visit,U} \bar{\theta}_{p,U} \quad \text{Eq. 3-22}$$

θ_p	Software unreliability estimated by feasible flowpath coverage based reliability estimation approach
$p_{visit,T}$	Probability that one of the tested flowpaths is visited on next execution
$p_{visit,U}$	Probability that one of the untested flowpaths is visited on next execution
$\bar{\theta}_{p,T}$	Mean unreliability of the tested flowpaths
$\bar{\theta}_{p,U}$	Mean unreliability of the untested flowpaths

The mean unreliability of the tested flowpaths is defined as the probability that an error is activated if one of the tested flowpath is chosen upon next execution. It is a little different from the probability that there exists an error on the next chosen flowpath if this flowpath has been tested before. The mean unreliability definition for the untested flowpaths is the same. It is the probability that we will observe a failure instead of the probability an error existing that matters in this study.

Before we finish a complete flowpath coverage testing, in the sense that every feasible flowpath has been tested at least once, we are not able to calculate either $p_{visit,T}$ or $p_{visit,U}$. The same obstacle needs to be overcome here as in the complete flowpath coverage testing task. Theoretically we can count how many flowpaths exist from the flowgraph of a program. We may never reach some of them due to the relationship among input data. Thus when talking about complete path coverage testing, instead of saying all the paths should be tested, people always put it as testing every possible path through out the flowgraph. As there is not any analytical solution to this problem yet, we will provide two strategies to estimate total number of possible flowpaths, the probability that one of the tested flowpaths is going to be visited, etc. based on the experimental results obtained through the testing process.

3.3.3.2.3. Estimate the Average Unreliability of the Tested Flowpaths

This estimation is carried out in exactly the same way as what is done to estimate the overall software reliability in the nodal coverage based reliability estimation approach. Every flowpath in this group has been tested as least once, so that we can take its visiting frequency

during testing as its relative visiting frequency within the group under real operating environment. The estimation of unreliability of every tested flowpath depend on the percentage of type I and type II errors and the probability that a type II error can be identified during a single test to that flowpath. The average unreliability of the tested flowpaths is:

$$\theta_{p,T} = \frac{\sum_{i_p=1}^{N_{p,T}} p_{visit,i_p} \hat{\theta}_{i_p}}{\sum_{i_p=1}^{N_{p,T}} p_{visit,i_p}} \quad \text{Eq. 3-23}$$

$\theta_{p,T}$	Average unreliability of the tested flowpaths
$N_{p,T}$	Total number of tested flowpaths in the program
p_{visit,i_p}	Conditional probability of flowpath i_p being visited during real operation given that one of the tested flowpaths is visited.
$\hat{\theta}_{i_p}$	Unreliability of flowpath i_p , estimated by Bayesian updating method

According to the first assumption in this methodology, we are performing an operational testing, that is, the sequence of test cases used are with the same statistical properties as those would be experienced in real operational use. Thus p_{visit,i_p} is simply approximated by the visiting frequency of flowpath i_p during testing.

$$p_{visit,i_p} = \frac{t_{visit,i_p}}{\sum_{i_p=1}^{N_{p,T}} t_{visit,i_p}} \quad \text{Eq. 3-24}$$

t_{visit,i_p}	Number of visits to flowpath i_p
-----------------	------------------------------------

3.3.3.2.3.1. Two Types of Software Defects

Software defects occur all the way through the life cycle of software development — from conception of product to end of life. The vernacular of development organizations tends to name and treat defects as different objects depending on when or where they are found. Some of the more common names are bug, error, comment, program trouble memoranda, problem, and authorized program analysis report (APAR)[Lyu96]. When a customer calls with a problem experienced with a product, it might be due to a software failure caused by a fault. On the other hand, not all problems experienced are due to the classical software-programming bug causing a failure. More often than not, a customer calls experiencing difficulties due to poor procedures, unclear documentation, poor user interfaces, etc.

So what is the definition of defect we shall use in this study and how do we categorize the type of defects? To answer this question, we have to make it clear what is our purpose of making

such definition and classification. We want to define and classify the defects we experienced during our testing process in a way such that the reliability of the software can be more precisely and easily estimated. As make this clear, we will simply define a failure in a traditional way, that is, “a failure is a deviation of the delivered service from compliance with the specification”. To make it more straightforward, in the context of this study, a failure is an output deviation of the delivered service from the output given by the oracle program with the same input data. Thus, the specification here is a broader concept than the actually plain language written design document. As failure is defined, a defect is the cause in the software product that triggers a failure. In this thesis, error, fault and defect will be used interchangeably. It is apparent that a defect can cause more than one failures if we do not correct it. It is also true that a failure can be caused by several defects. A failure is a view of a defect or several defects and a defect is a cause of a group of failures if no changes are made to correct those defects. If we are always able to correct any defects behind any observed failures, one defect can only cause one failure and we will not be able to observe it more than once. This situation is true in the tested flowpaths. As we have emphasized before, it is required that all known software defects are removed in the case of safety-critical software, as used in nuclear power plants. In the testing process discussed in the previous section, it is

Now it comes to the question how we classify the problems experienced during testing process. The method should be able to help us estimate the software product reliability. This objective leads us to the resulting defects classification technique adopted in this study: type I error / defect / fault — the probability of encountering such an error during the first execution of its host flowpath is equal to or very close to unity and type II error / defect / fault — the probability of encountering such an error during any single visit to its host flowpath is the same and independent to all the others. There is one point that needs to be emphasized, i.e., both types of errors are associated with certain flowpaths instead of certain nodes. If the errors are believed to connect with nodes, very few errors can be encountered during the first visit to a node during the integrated testing process since all of the nodes should have passed some level of modular testing. Here, both types of defects are believed to relate to some data flow that bypass their control flows — flowpaths. For the type I defects, almost all the input data, which lead to the flowpath that the error stays, can trigger that error. For the type II defects, only part of the input data that lead to the flowpath can trigger the error. Because the input data are sampled randomly, both the input data that can trigger the error and those that cannot trigger the error have non-zero opportunities to be chosen during any single sampling process to that flowpath. The possibilities are depending on the fraction of the data that can activate the error and the fraction that cannot activate the error out of all the input data that can lead to that flowpath. Since each input data sampling process is independent to all the others and there are only two results, the input data belong to the error-triggering group or the non-error triggering group, if the input data can lead to the execution of a flowpath where a type II error stays, each process is an independent Bernoulli process. An obvious example of the type II error is the boundary-condition error, which is only activated when certain boundary condition is satisfied. Even though there are lots of other input data that can lead to execution of the same flowpath, they are not able to help the tester to detect the boundary type errors.

In order to incorporate this two-error-type scenario into our reliability estimation process, two important probability values should be available. The first one is the probability that an error chosen error is of type I if it is randomly chosen from a pool of errors from the target software or similar software systems. In another word, the first value we need to know about this error category scenario is the percentage of errors or defects that belong to error type I. Let us denote this probability as p_I . It is obvious that the probability that a randomly chosen error belongs to the second type, $p_{II} = 1 - p_I$. The second value we need to know is the probability that we will encounter a type II error on next execution of the flowpath if there is a type II error on that

flowpath. This probability should be the same for every single visit to a flowpath given that flowpath has a type II error since each input data set is chosen independently in the same manner. Let us denote this probability as $p_{f,II}$. For type I error, according to the error classification mechanism, the probability of finding a type I error during the first visit to a flowpath, given there is a type I error on that flowpath, is $p_{f,I}^1 = 1$. Since it is required that all the errors be removed, we will not be able to meet this error during another further visits to the same flowpath, which means $p_{f,I}^{1+} = 0$. It is true that p_I , p_{II} and $p_{f,II}$ are different for different programs, even different flowpaths in the same software system. But since the average unreliability of the software is used as an estimation, p_I , p_{II} and $p_{f,II}$ can be taken as constants in the average sense. Otherwise stated, the average p_I , p_{II} and $p_{f,II}$ are constants in the same program or similar programs. Thus only the failure data in the same program or similar programs are used to update the probability distribution of p_I and $p_{f,II}$ according to Bayesian framework. With the distributions on hand, the mean values or median values could be applied to estimate reliabilities / unreliabilities of the flowpaths, hence estimate the reliability of the software.

3.3.3.2.3.2. Update Probability Distributions Associated with The Two Error Types

In this section, a brief review on Bayesian updating method is presented first, followed with two approaches to update probability distributions of p_I , the fraction of errors that are of type I, and $p_{f,II}$, the probability of encountering a type II error during a single test on a flowpath given there is a type II error on that flowpath.

Before Bayesian method, there are several classical statistical approaches, e.g., point and interval estimation, etc. to estimate the distribution parameters. These approaches assume that the parameters are constants (but unknown) and that the sample statistics are used as estimators of these parameters. Because the estimators are invariably imperfect, errors of estimation are unavoidable. In the classical approaches, confidence intervals are used to express the degree of these errors. Accurate estimates of parameters in these approaches require large amount of data. The Bayesian framework addresses estimation problem from another point of view. In this case, the unknown parameters of a distribution are assumed (or modeled) to be also random variables. In this way, uncertainty associated with the estimation of the parameters can be combined formally through Bayes' theorem with the inherent variability of the basic random variable. With this approach, subjective judgments, based on intuition, experience, or indirect information are incorporated systematically with observed data to obtain a balanced estimation. The Bayesian method is particularly helpful in cases where there is a strong basis for such judgments as well as where the available data are sparse.

The Bayes' theorem is as follows. Consider n mutually exclusive and collectively exhaustive events $E_1, E_2, \dots, E_n$, that is, $E_1 \cup E_2 \cup \dots \cup E_n = S$. Then if A is an event also in the same sample space (see the following picture), we have:

$$P(E_i | A) = \frac{P(A | E_i)P(E_i)}{P(A)} = \frac{P(A | E_i)P(E_i)}{\sum_{j=1}^n P(A | E_j)P(E_j)} \quad \text{Eq. 3-25}$$

Figure 3-3 Venn diagram with events A and $E_1, E_2, \dots, E_n$

Bayesian updating method is based on Bayes' theorem. Suppose that the possible values of a parameter θ are assumed to be a set of discrete values $\theta_i, i = 1, 2, \dots, n$, with relative likelihoods $p_i = P(\Theta = \theta_i)$. Θ is the random variable whose values represent possible values of the parameter θ . If additional information becomes available (such as the results of a series of tests or experiments), the prior assumptions on the parameter θ may be modified by Bayes' theorem as follows.

If we denote ε as the observed outcome of the experiment, then applying Bayes' theorem, we obtain the updated PMF of Θ as:

$$P(\Theta = \theta_i | \varepsilon) = \frac{P(\varepsilon | \Theta = \theta_i)P(\Theta = \theta_i)}{\sum_{j=1}^n P(\varepsilon | \Theta = \theta_j)P(\Theta = \theta_j)}, i = 1, 2, \dots, n \quad \text{Eq. 3-26}$$

The terms in the above formula are interpreted as:

$P(\varepsilon \Theta = \theta_i)$	The likelihood of the experiment outcome ε if $\Theta = \theta_i$; that is the conditional probability of obtaining a particular experimental outcome assuming that the parameter has value θ_i
$P(\Theta = \theta_i)$	The prior probability of $\Theta = \theta_i$; that is, prior to the availability of the experimental information ε
$P(\Theta = \theta_i \varepsilon)$	The posterior probability of $\Theta = \theta_i$; that is, the probability that has been revised in the light of the experimental outcome ε

If we denote the prior and posterior probabilities as $P'(\Theta = \theta_i)$ and $P''(\Theta = \theta_i)$ respectively, we have:

$$P''(\Theta = \theta_i) = \frac{P(\varepsilon | \Theta = \theta_i)P'(\Theta = \theta_i)}{\sum_{j=1}^n P(\varepsilon | \Theta = \theta_j)P'(\Theta = \theta_j)} \quad \text{Eq. 3-27}$$

The expected value of Θ is commonly used as the Bayesian estimator of the parameter, that is,

$$\hat{\theta}'' = E(\Theta | \varepsilon) = \sum_{j=1}^n \theta_j P''(\Theta = \theta_j) \quad \text{Eq. 3-28}$$

The continuous formulas for Bayesian method are:

$$f''(\theta) = \frac{P(\varepsilon | \theta) f'(\theta)}{\int_{-\infty}^{+\infty} P(\varepsilon | \theta) f'(\theta) d\theta} \quad \text{Eq. 3-29}$$

$$\hat{\theta}'' = \int_{-\infty}^{+\infty} \theta f''(\theta) d\theta \quad \text{Eq. 3-30}$$

Based on our knowledge of the nature of each failure encountered during testing process, there are two approaches to update the two probability values we are interested in.

If we are able to distinguish the type I and type II errors, that is, divide all the encountered errors into the two error groups, we can update p_I and $p_{f,II}$ separately. If an error is detected during the first visit to a flowpath, it is either a type I error or a type II error because the type I error is always detected during the first visit to a flowpath and the type II error also has some opportunity ($p_{f,II}$) to be detected during the first visit. Thus for all the errors detected during the first visit to a flowpath, the physics behind the error should be examined; categorization is made manually. If an error is detected during a later test to the host flowpath, it is a type II error. After separating the errors detected on the first visits, the number of visits to a flowpath that it takes to detect an error is the only information or fact we need to use to update p_I , p_{II} and $p_{f,II}$. The steps are described as follows.

Let's assume we only use error data from testing result of the program under testing to update the error-related probabilities. First, use all available error data to update p_I . Assume the prior probability distribution of p_I is $f_0(p_I)$. If there is no prior knowledge about p_I , we can use the ignorant uniform probability distribution:

$$f_0(p_I) = 1$$

If the first encountered error is of type I, then $f_0(p_I)$ is updated as:

$$f_1(p_I) = C_1 p_I f_0(p_I) \quad \text{Eq. 3-31}$$

This is because given p_I , the conditional probability that an error is of type I is p_I . C_1 is the normalization constant that can be obtained by:

$$\begin{aligned} \int_0^1 f_1(p_I) dp_I &= \int_0^1 C_1 p_I f_0(p_I) dp_I = 1 \\ \Rightarrow C_1 &= \frac{1}{\int_0^1 p_I f_0(p_I) dp_I} \end{aligned} \quad \text{Eq. 3-32}$$

If the second error encountered is of type II, the probability density function of p_I can be further updated as:

$$f_2(p_I) = C'_2(1 - p_I)f_1(p_I) = C_2 p_I(1 - p_I)f_0(p_I) \quad \text{Eq. 3-33}$$

$$C_2 = \frac{1}{\int_0^1 p_I(1 - p_I)f_0(p_I)dp_I}$$

In general,

$$f_s(p_I) = \begin{cases} C_s p_I f_{s-1}(p_I) & C_s = \frac{1}{\int_0^1 p_I f_{s-1}(p_I) dp_I} \quad \text{The } s^{th} \text{ error is of type I} \\ C_s (1 - p_I) f_{s-1}(p_I) & C_s = \frac{1}{\int_0^1 (1 - p_I) f_{s-1}(p_I) dp_I} \quad \text{The } s^{th} \text{ error is of type II} \end{cases}$$

Eq. 3-34

Notice that we have been taking $[0, 1]$ as the integral range of p_I . This is because p_I is a probability, which implies it is equal or greater than zero and equal or less than one. After all the error data have been used to update probability density function of p_I , that is, $f_s(p_I)$, the mean value of p_I can be used as the estimator of the fraction of errors that belong to the first category:

$$\hat{p}_I = \int_0^1 p_I f_s(p_I) dp_I, \quad \text{Eq. 3-35}$$

where N_e is the total number of errors detected on the program. And the fraction of errors that belong to the second category, p_{II} , is estimated as:

$$\hat{p}_{II} = 1 - \hat{p}_I \quad \text{Eq. 3-36}$$

Now we need to update probability that a type II error is encountered during any single visit to a flowpath, $p_{f,II}$. As mentioned before, this probability should be different for different flowpaths and different errors; even they are within the same program. But since only the average effect matters to us, the average probability can be looked as a constant and updated based on Bayesian method. Use all available type II error data to update $p_{f,II}$. Assume the prior probability distribution of $p_{f,II}$ is $f_0(p_{f,II})$. If there is no prior knowledge about $p_{f,II}$, we can use the ignorant uniform probability distribution:

$$f_0(p_{f,II}) = 1 \quad \text{Eq. 3-37}$$

If the first type II error is detected on the 4th visit to a flowpath, then $f_0(p_{f,II})$ is updated as:

$$f_1(p_{f,II}) = C_1(1 - p_{f,II})^3 p_{f,II} f_0(p_{f,II})$$

$$C_1 = \frac{1}{\int_0^1 (1 - p_{f,II})^3 p_{f,II} f_0(p_{f,II}) dp_{f,II}} \quad \text{Eq. 3-38}$$

This is because given $p_{f,II}$, the probability that a type II error is detected on the 4th visit to a flowpath is the product of the probability that the error is not detected during the first three visits, which is $(1 - p_{f,II})^3$ and the probability that the error is detected on the 4th visit, which is $p_{f,II}$.

If the second type II error is detected on the 2nd visit to a flowpath, then $f_1(p_{f,II})$ is further updated as:

$$f_2(p_{f,II}) = C'_2(1 - p_{f,II}) p_{f,II} f_1(p_{f,II}) = C_2(1 - p_{f,II})^4 p_{f,II}^2 f_0(p_{f,II})$$

$$C_2 = \frac{1}{\int_0^1 (1 - p_{f,II})^4 p_{f,II}^2 f_0(p_{f,II}) dp_{f,II}} \quad \text{Eq. 3-39}$$

If the s^{th} type II error is detected during the k_s^{th} visit to the flowpath, the probability density function of $p_{f,II}$ is updated the s^{th} times as:

$$f_s(p_{f,II}) = C_s(1 - p_{f,II})^{k_s-1} p_{f,II} f_{s-1}(p_{f,II})$$

$$C_s = \frac{1}{\int_0^1 (1 - p_{f,II})^{k_s-1} p_{f,II} f_{s-1}(p_{f,II}) dp_{f,II}} \quad \text{Eq. 3-40}$$

After all the type II error data have been used to update probability density function of $p_{f,II}$, that is, $f_{N_{e,II}}(p_{f,II})$, the mean value of $p_{f,II}$ can be used as the estimator of the average probability that a type II error is detected on a single visit to the error hosting flowpath, that is:

$$\hat{p}_{f,II} = \int_0^1 p_{f,II} f_{N_{e,II}}(p_{f,II}) dp_{f,II} \quad \text{Eq. 3-41}$$

where $N_{e,II}$ is the total number of type II errors detected on the program.

What should we do if we could not separate the two groups of errors? This situation is not very unusual since very detailed knowledge about the program is needed in order to separate them and sometimes, even with all the possible information about the error on hand, it is still hard to tell whether an error detected on the first visit to a flowpath is of type I or type II. The answer to this question is rather simple, that is to update the joint probability distribution function of p_I and $p_{f,II}$, considering they are independent to each other. It is obvious that the fraction of type I error has nothing to do with the probability of detecting a type II error during a single visit to a flowpath that has a type II error. Now the only error data we need are how many number of visits it takes to detect an error.

The prior joint distribution of p_I and $p_{f,II}$ is:

$$f_0(p_I, p_{f,II}) = f_0(p_I)f_0(p_{f,II}), 0 \leq p_I \leq 1, 0 \leq p_{f,II} \leq 1 \quad \text{Eq. 3-42}$$

If we do not have any knowledge about p_I and $p_{f,II}$ (may obtained from other similar programs), we can take ignorant prior probability distributions for all of them, that is,

$$f_0(p_I) = f_0(p_{f,II}) = f_0(p_I, p_{f,II}) \quad \text{Eq. 3-43}$$

If the first error is detected during the 2nd visit to a flowpath, $f_0(p_I, p_{f,II})$ is updated as:

$$f_1(p_I, p_{f,II}) = C_1(1 - p_I)(1 - p_{f,II})p_{f,II}f_0(p_I, p_{f,II}) \quad \text{Eq. 3-44}$$

$$C_1 = \frac{1}{\iint_{p_I, p_{f,II}} (1 - p_I)(1 - p_{f,II})p_{f,II}f_0(p_I, p_{f,II})dp_Idp_{f,II}}$$

The probability that an error is encountered during the 2nd visit to a flowpath is the sum of the probability that it is a type I error and detected during the 2nd visit to a flowpath and the probability that it is a type II error and detected during the 2nd visit. The first probability is 0 due to the definition of the type I error. The 2nd probability is product of the probability that this error is of type II and the probability of finding the error on the 2nd visit given it is a type II error.

If the second error is detected during the 1st visit to a flowpath, $f_1(p_I, p_{f,II})$ is once more updated as:

$$f_2(p_I, p_{f,II}) = C'_2[p_I + (1 - p_I)p_{f,II}]f_1(p_I, p_{f,II})$$

$$f_2(p_I, p_{f,II}) = C_2[p_I + (1 - p_I)p_{f,II}](1 - p_I)(1 - p_{f,II})p_{f,II}f_0(p_I, p_{f,II}) \quad \text{Eq. 3-45}$$

$$C_2 = \frac{1}{\iint_{p_I, p_{f,II}} [p_I + (1 - p_I)p_{f,II}](1 - p_I)(1 - p_{f,II})p_{f,II}f_0(p_I, p_{f,II})dp_Idp_{f,II}}$$

In general, if s^{th} error is detected during the 1st visit to a flowpath, the updated joint probability density function of p_I and $p_{f,II}$ is:

$$f_s(p_I, p_{f,II}) = C_s [p_I + (1 - p_I)p_{f,II}] f_{s-1}(p_I, p_{f,II})$$

$$C_s = \frac{1}{\iint_{p_I, p_{f,II}} f_s(p_I, p_{f,II}) dp_I dp_{f,II}}$$
Eq. 3-46

If s^{th} error is detected during the k_s^{th} visit to a flowpath ($k_s > 1$), the updated joint probability density function of p_I and $p_{f,II}$ is:

$$f_s(p_I, p_{f,II}) = C_s (1 - p_I)(1 - p_{f,II})^{k_s-1} p_{f,II} f_{s-1}(p_I, p_{f,II})$$

$$C_s = \frac{1}{\iint_{p_I, p_{f,II}} f_s(p_I, p_{f,II}) dp_I dp_{f,II}}$$
Eq. 3-47

The mean value of p_I serves as the estimator of p_I as:

$$\hat{p}_I = \iint_{p_I, p_{f,II}} f_{N_e}(p_I, p_{f,II}) p_I dp_I dp_{f,II}$$
Eq. 3-48

And the mean value of $p_{f,II}$ is utilized as the estimator of $p_{f,II}$ as:

$$\hat{p}_{f,II} = \iint_{p_I, p_{f,II}} f_{N_e}(p_I, p_{f,II}) p_{f,II} dp_I dp_{f,II},$$
Eq. 3-49

where N_e is the total number of errors encountered during test.

3.3.3.2.3.3. Estimate Reliability of a Tested Flowpath

Now, we have categorized the errors in a software program into two groups. With the estimation of p_I , p_{II} and $p_{f,II}$, we are able to estimate unreliability of every single tested flowpath. So, what is the unreliability of a flowpath? The unreliability of flowpath i_p is the failure probability if flowpath i_p is executed. Then what is the fact that we can use to estimate or update unreliability of flowpath i_p , θ_{i_p} ? It is the same form for every tested flowpath, that is, the flowpath has been tested t_{i_p} times without any error. This is always true for any tested flowpath since it is required that all known faults in the software program are removed.

Now, let's estimate the unreliability of a flowpath, θ , given it has been tested t times without an error. It equals the probability that there is an error on the flowpath times the probability that the error is met upon next execution, that is,

$$\theta = p_e \times p_f \quad \text{Eq. 3-50}$$

p_e	The probability that there is an error on the flowpath
p_f	The probability that the error is encountered upon next execution given there is an error on the flowpath

From above formula, it seems the probability that there are more than one error on one flowpath is ignored. This is not the exact case. While updating the probability density functions of p_e and $p_{f,II}$, if there are more than one error on the same flowpath, they will be treated as multiple errors on multiple flowpaths, which makes the calculations simpler and maintains the result unreliability estimates at the right level (correctly indicating the testing results). Let's look at p_e and p_f separately.

There are two ways to estimate p_e , the probability that there is an error on a flowpath based on the fact that it has been tested t time error free. The first method is to assume p_e has a continuous probability distribution between 0 and 1. If we have tested a flowpath without any error, according to Bayesian updating framework, p_e is the conditional probability that there is an error on the flowpath given that we have tested the flowpath once without any error.

$$f_1(p_e) = P(p_e | \bar{F}) = \frac{P(p_e, \bar{F})}{P(\bar{F})} = \frac{P(\bar{F} | p_e) f_0(p_e)}{\int_0^1 P(\bar{F} | p_e) f_0(p_e) dp_e}, \quad 0 \leq p_e \leq 1 \quad \text{Eq. 3-51}$$

$$f_1(p_e) = \frac{[P(\bar{F}, E | p_e) + P(\bar{F}, \bar{E} | p_e)] f_0(p_e)}{\int_0^1 [P(\bar{F}, E | p_e) + P(\bar{F}, \bar{E} | p_e)] f_0(p_e) dp_e} \quad \text{Eq. 3-52}$$

$$f_1(p_e) = \frac{[P(\bar{F} | E, p_e) P(E | p_e) + P(\bar{F} | \bar{E}, p_e) P(\bar{E} | p_e)] f_0(p_e)}{\int_0^1 [P(\bar{F} | E, p_e) P(E | p_e) + P(\bar{F} | \bar{E}, p_e) P(\bar{E} | p_e)] f_0(p_e) dp_e} \quad \text{Eq. 3-53}$$

$\bar{F}$	The flowpath has been tested once without any error
E	There is an error on the flowpath
$\bar{E}$	There is no error on the flowpath
p_e	The probability that there is an error on the flowpath
$f_0(p_e)$	Prior probability density function of p_e
$f_1(p_e)$	Posterior probability density function of p_e

The probability that we do not observe a failure given there is an error on a flowpath is the sum of the probability that we do not observe a failure and there is a type I error on the flowpath given there is an error on the flowpath and the probability that we do not observe a failure and there is a type II error on the flowpath given there is an error on the flowpath, which can be written as $P(\bar{F} | E, p_e) = P(\bar{F} | E_I, E, p_e)P(E_I | E, p_e) + P(\bar{F} | E_{II}, E, p_e)P(E_{II} | E, p_e)$.

$P(\bar{F} | E_I, E, p_e)$ is the conditional probability that we do not meet a failure during a test given there is a type I and only a type I error on the flowpath. According to the definition of type I error, this is equal to zero since a type I error is always found during the first visit to a flowpath. Otherwise stated, if there is an uncorrected type I error on a flowpath, the flowpath always fails during an execution. Thus, $P(\bar{F} | E_I, E, p_e) = 0$.

$P(E_I | E, p_e)$ is the probability that a random selected error belongs to type I, or, the fraction of type I error among all errors. This is exactly the definition of p_I . Thus, $P(E_I | E, p_e) = p_I$.

$P(\bar{F} | E_{II}, E, p_e)$ is the conditional probability that we do not meet a failure during a test given there is a type II and only a type II error on the flowpath. According to definition of $p_{f,II}$, the probability that a type II error is encountered during one test, the probability that a type II error is not encountered is $(1 - p_{f,II})$. Thus, $P(\bar{F} | E_{II}, E, p_e) = (1 - p_{f,II})$.

$P(E_{II} | E, p_e)$ is the probability that a random selected error belongs to type II, or, the fraction of type II error amount all errors. Since we only have two types of errors, we have $P(E_{II} | E, p_e) = p_{II} = 1 - p_I$.

To sum up, we have:

$$P(\bar{F} | E, p_e) = (1 - p_{f,II})(1 - p_I) \quad \text{Eq. 3-54}$$

It is apparent that if there is no error on the flowpath, we cannot meet any error during any test. Thus, we have:

$$P(\bar{F} | \bar{E}, p_e) = 1 \quad \text{Eq. 3-55}$$

Also, $P(E | p_e) = p_e$ and $P(\bar{E} | p_e) = 1 - p_e$. Thus, one time update of the probability density distribution of p_e can be written as:

$$f_1(p_e) = C_1[p_e(1 - p_I)(1 - p_{f,II}) + (1 - p_e)]f_0(p_e) \\ C_1 = \int_0^1 [p_e(1 - p_I)(1 - p_{f,II}) + (1 - p_e)]f_0(p_e)dp_e, \quad 0 \leq p_e \leq 1 \quad \text{Eq. 3-56}$$

After the second test, still without any error, the probability density function is updated again as:

$$\begin{aligned}
f_2(p_e) &= C_2'[p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]f_1(p_e) \\
f_2(p_e) &= C_2[p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^2 f_0(p_e), \quad 0 \leq p_e \leq 1 \\
C_2 &= \int_0^1 [p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^2 f_0(p_e) dp_e
\end{aligned} \tag{Eq. 3-57}$$

After t tests without any failure, the probability density function is updated as:

$$\begin{aligned}
f_t(p_e) &= C_t[p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^t f_0(p_e) \\
C_t &= \int_0^1 [p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^t f_0(p_e) dp_e, \quad 0 \leq p_e \leq 1
\end{aligned} \tag{Eq. 3-58}$$

The average of p_e for every flowpath is used as the estimator of the error existing probability of that flowpath. Thus, for a flowpath we have tested t times without any error (this is the case for all the tested flowpaths due to the perfect error removal requirement), its probability of containing an error is estimated as:

$$\begin{aligned}
\hat{p}_e &= C_t \int_0^1 [p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^t p_e f_0(p_e) dp_e \\
C_t &= \int_0^1 [p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^t f_0(p_e) dp_e
\end{aligned} \tag{Eq. 3-59}$$

$f_0(p_e)$ is chosen based on our prior confidence about the program before any test has been performed.

Sometimes, the above discussed Bayesian updating technique could be very difficult to perform depending on the form of prior distribution $f_0(p_e)$. Therefore, we want to introduce a simplified method of estimating $\hat{p}_e$. This time, let us assume p_e is a value, not as in Bayesian updating, where it has a distribution. After we have performed t tests on a flowpath without any error, p_e can be written as:

$$p_e' = P(E | \bar{F}^t) = \frac{P(E, \bar{F}^t)}{P(\bar{F}^t)} = \frac{P(\bar{F}^t | E)P(E)}{P(\bar{F}^t | E)P(E) + P(\bar{F}^t | \bar{E})P(\bar{E})} \tag{Eq. 3-60}$$

$\bar{F}^t$	The flowpath has been tested t times without any error
E	There is an error on the flowpath
$\bar{E}$	There is no error on the flowpath
p_e'	The probability that there is an error on the flowpath after t error free test on the flowpath

As discussed above,

$$P(\bar{F}^t | E) = (1 - p_I)(1 - p_{f,II})^t \quad \text{Eq. 3-61}$$

$$P(\bar{F}^t | \bar{E}) = 1$$

$$P(E) = p_e^0$$

$$P(\bar{E}) = 1 - P(E) = 1 - p_e^0$$

p_e^0 is the prior value of the probability that there is an error on the flowpath. It is obvious that the mean value of $f_0(p_e)$ used in the previous Bayesian updating method should be used as p_e^0 , although the resulting p_e^t is not necessarily the same as $\hat{p}_e$. To sum up, this simpler method to estimate p_e is:

$$p_e^t = \frac{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0}{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0 + (1 - p_e^0)} \quad \text{Eq. 3-62}$$

For both methods, the estimators, $\hat{p}_I$ and $\hat{p}_{f,II}$ obtained from the error data of the same program or similar programs are plugged into their respective formula to get the error existing probability for each tested flowpath.

In order to estimate unreliability of a tested flowpath, θ , besides p_e , which can be

$$\hat{p}_e = C_t \int_0^1 [p_e(1 - p_I)(1 - p_{f,II}) + (1 - p_e)]^t p_e f_0(p_e) dp_e$$

solved either by $C_t = \int_0^1 [p_e(1 - p_I)(1 - p_{f,II}) + (1 - p_e)]^t f_0(p_e) dp_e$ Eq. 3-59, or

$$p_e^t = \frac{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0}{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0 + (1 - p_e^0)}$$

Eq. 3-62, we have to solve p_f , the probability a

failure will be encountered upon next execution on the flowpath given there is an error on the flowpath according to $\theta = p_e \times p_f$ Eq. 3-50. Here another assumption is made. Different from p_e , which is unique for every tested flowpath, an average value is used for p_f of every flowpath. This is because we have sparse error data from the software we are testing and is a natural extension of the single value assumption for $p_{f,II}$, which will become very clear after the method to estimate p_f is discussed. Let us use divide and conquer strategy to estimate p_f , which can be divided into two parts, the probability that we will detect an error upon next execution if there is a type I error on the flowpath, $P(F | E_I)$ and the same probability if there is a type II error on the flowpath, $P(F | E_{II})$. Following this logic, p_f can be written as:

$$p_f = P(F | E_I)p_I^t + P(F | E_{II})p_{II}^t \quad \text{Eq. 3-63}$$

p_f

The probability that a failure is encountered upon next execution given there is an error on the flowpath

	error on the flowpath
$P(F E_I)$	The probability that a failure is encountered upon next execution given there is a type I error on the flowpath
p_I^t	The probability that an error is of type I, given it has been tested t times failure free, $t \geq 1$
$P(F E_{II})$	The probability that a failure is encountered upon next execution given there is a type II error on the flowpath
p_{II}^t	The probability that an error is of type II, given it has been tested t times failure free, $t \geq 1$

We need to be aware of the fact that the flowpaths we are dealing with now are those flowpaths that have been tested at least once without any failure. Were there a type I error on any of these flowpaths, it should have been encountered during the first visit on it according to the definition of type I error. This is to say $p_I^t = 0$. Thus there is no need to calculate $P(F | E_I)$ for the purpose of estimating p_f . Also, as $p_I^t + p_{II}^t = 1$, since there are only two types of errors, $p_{II}^t = 1 - p_I^t = 1$. The last term is $P(F | E_{II})$, which is exactly the definition of $p_{f,II}$, the probability that an type II error is encountered upon a single execution given there is a type II error on the visited flowpath. Plug all the four terms into $p_f = P(F | E_I)p_I^t + P(F | E_{II})p_{II}^t$ Eq. 3-63, we get:

$$p_f = p_{f,II} \quad \text{Eq. 3-64}$$

3.3.3.2.3.4. Summary — Estimate Unreliability of tested flowpaths

$$\theta_{p,T} = \frac{\sum_{i_p=1}^{N_{p,T}} p_{visit,i_p} \hat{\theta}_{i_p}}{\sum_{i_p=1}^{N_{p,T}} p_{visit,i_p}}$$

According to Eq. 3-23, the unreliability of the tested part of the software is a weighted average over unreliability of all the tested flowpath, with visiting frequency to each flowpath during testing as its weight:

$$\theta_{p,T} = \frac{\sum_{i_p=1}^{N_{p,T}} p_{visit,i_p} \hat{\theta}_{i_p}}{\sum_{i_p=1}^{N_{p,T}} p_{visit,i_p}}$$

According to $\theta = p_e \times p_f$ Eq. 3-50, unreliability of each tested flowpath is estimated as:

$$\theta = p_e \times p_f$$

The probability that there is an error in the flowpath, p_e , is unique for each flowpath and can be

$$\hat{p}_e = C_t \int_0^1 [p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^t p_e f_0(p_e) dp_e$$

estimated by $C_t = \int_0^1 [p_e(1-p_I)(1-p_{f,II}) + (1-p_e)]^t f_0(p_e) dp_e$ Eq. **3-59**,

assuming continuous distribution for p_e or $p_e^t = \frac{(1-\hat{p}_I)(1-\hat{p}_{f,II})^t p_e^0}{(1-\hat{p}_I)(1-\hat{p}_{f,II})^t p_e^0 + (1-p_e^0)}$ Eq. **3-62**,

only caring about the average value of p_e . The probability that a failure is encountered during next execution on a flowpath given there is an error on the flowpath, p_f is equal to the probability that a failure is encountered during next execution on a flowpath given there is a type II error on the flowpath in light of the fact that none of the tested flowpath can be any type I error, i.e. $p_f = p_{f,II}$.

During the above mentioned calculation, the two-type-error model is used. The relative probabilities, p_I , the probability that an error is of type I and $p_{f,II}$, the probability that a failure is encountered during a single visit to a flowpath due to a type II error can either be estimated separately or jointly. If the tester is able to separate all the encountered errors during testing into the two groups, separate distribution updating can be performed according to Eq.

$$\hat{p}_I = \int_0^1 p_I f_s(p_I) dp_I, \quad \text{Eq. 3-35, } \hat{p}_{II} = 1 - \hat{p}_I \quad \text{Eq. 3-36,}$$

$$f_s(p_{f,II}) = C_s (1-p_{f,II})^{k_s-1} p_{f,II} f_{s-1}(p_{f,II})$$

$$C_s = \frac{1}{\int_0^1 (1-p_{f,II})^{k-1} p_{f,II} f_{s-1}(p_{f,II}) dp_{f,II}} \quad \text{Eq. 3-40 and } \hat{p}_{f,II} = \int_0^1 p_{f,II} f_{N_e,II}(p_{f,II}) dp_{f,II}$$

Eq. **3-41**, which is easier in terms of calculation. If the tester cannot tell type I errors from type II errors, a joint probability distribution of p_I and $p_{II,f}$ can be carried out and

$$f_s(p_I, p_{f,II}) = C_s (1-p_I)(1-p_{f,II})^{k_s-1} p_{f,II} f_{s-1}(p_I, p_{f,II})$$

$$C_s = \frac{1}{\iint_{p_I, p_{f,II}} f_s(p_I, p_{f,II}) dp_I dp_{f,II}}$$

estimations obtained according to

$$\hat{p}_{f,II} = \iint_{p_I, p_{f,II}} f_{N_e}(p_I, p_{f,II}) p_{f,II} dp_I dp_{f,II}, \quad \text{Eq. 3-47, Eq. 3-49.}$$

3.3.3.2.4. Estimate the Average Unreliability of the Untested Flowpaths

For most of the software programs, even the fully testable and substantially testable ones, it is seldom the case that all the possible flowpaths are tested. Furthermore, we even do not have a method to count how many possible flowpaths are in a program (This is discussed in detail in the next section). The only information we get after the testing is from the tested flowpaths. Our task now is to estimate the unreliability of the untested flowpaths based on the information we obtained from the tested flowpaths. To complete this task, moving from the information of the tested flowpaths to the unreliability of the untested flowpaths, we have to know the difference between these two parts of the same software as well as the similarities. We need to take

advantage of the common features shared by them and bridge the differences. Thus, in this section, we start with these common or different characters between the tested and untested part and continue with how to deal with them in order to get average unreliability estimation for the untested flowpaths from the testing information we have from the tested flowpaths.

3.3.3.2.4.1. Similarities And Differences between Tested and Untested Flowpaths

First, the tested and untested flowpaths belong to the same software program. They share lots of common nodes, all the nodes if complete nodal coverage testing has been performed on the underlying software. All of them are designed and programmed by the same group of people under the same development environment. It is reasonable to say they are at the same reliability level if neither of them has been tested and modified, that is the average reliability of the software before testing. Then how about the reliability if we test some parts of the software, say, the tested flowpaths, without any correction to any of the detected errors? Are they still the same? The answer is yes. It is always mistakenly thought that the reliability of a software program should be improved through testing the software. But it is not true. Only if the detected errors are removed, at least some of them are successfully and perfectly corrected, the reliability increases. This is because testing does not change the software if we do not modify it. The software is still the same after pure testing. The real effect of testing is not on the software itself. Rather, the testing process improves our knowledge about the software. If we were to update software reliability upon pure testing result, it is not because the underlying reliability has changed. Instead it is because we have more knowledge about the software and could get a more precise or close estimation of the software reliability. Thus, we can say the tested and untested software flowpaths are at the same reliability level before and after pure testing process. By pure testing, we mean testing without any modification. For the same reason, if both parts are under the same testing and correcting process, they should be at the same reliability level. In that case, we do not have two parts any more and all the flowpaths would have been tested and modified. All the flowpaths of the software become tested flowpaths.

But what if we do test and modify the program at the same time. As in this case, we test the software and always perfectly correct any error detected during testing. Obviously, we increase reliability of the tested flowpaths if we only care about the average reliability of the tested and untested flowpaths. Now we introduce a difference between tested and untested flowpaths in terms of reliability by modifying the tested flowpaths.

Now, we believe the reliability of the tested flowpaths is higher than the reliability of the untested flowpaths. But is the untested part as unreliability as the software before any test and modification is done on the software? Does the modification of the tested flowpaths improve the quality of the untested flowpaths? Yes, since they share lots of nodes. It is very possible that if an error on a node is corrected due to testing of a tested flowpath, some potential failure observations from the untested flowpath are removed too. In another word, due to the error on the same node shared by different flowpaths belonging to both the tested flowpaths and to the untested flowpaths, we will observe less failures on the untested flowpaths than if we do not make any modification on the tested flowpaths.

To sum up, the reliability of the tested and untested flowpaths are at the same level before testing and modification. If the same testing and modification process is performed on both parts of the software, they will be at the same reliability level. The fact that both parts of the software are designed and developed by the same group of people under the same environment is the reason for these similarities. But since we cannot manage complete flowpath coverage on the underlying software, the tested flowpaths are modified while the untested flowpaths have not been modified. This causes the difference between these two parts in terms of reliability. Then the question we need to answer is how big the difference is. The difference is smaller than the difference between the current reliability level of the tested and modified flowpaths and initial reliability level of the software without any modification due to the common errors on common

nodes shared by both parts of the software. This is the beneficial effect on the untested flowpaths from modifying the tested flowpaths.

3.3.3.2.4.2. Unreliability of Untested Flowpaths

There are two versions of reliabilities due to the difference between failures and errors. Before we get into how to estimate the reliability of the untested flowpaths, let us take one more look at which reliability is the one we are interested in. The similarities and differences described in the previous section then are used to estimate the desired reliability. Again, the unreliability is the feature we estimate directly. The reliability is simply one minus the unreliability.

It is very intuitive to see that one error may cause several failures if we do not perfectly correct it immediately. The reason is that there are normally more than one path sharing the faulty node, which therefore may cause more than one path to fail. Even though the error only affects one path, the error can still be triggered by different inputs that lead to the same faulty path. To put it in a simple way, a single faulty node may be associated with several faulty paths and a faulty path may be associated with several unique input data sets. In Figure 3-4, one faulty node, node 5 can cause two flowpaths, 1-2-5-10 and 1-2-5-11 to fail. If input data set 1, 2, 3, 4, 5 and 6 lead to either of these two flowpaths, at most six test cases may fail due to the error on node 5. From perspective of the program, only one error exists (assuming all the other nodes are correct for this example) while from perspective of failures observed, there might be six at most¹.

Figure 3-4 Six Failure Observations due to One Error

Now, let us define two terminologies, the intrinsic reliability and the observed reliability. The reliability estimated using the number of errors encountered during the testing and perfect correcting process is termed as the intrinsic reliability. The reliability estimated from the number of errors encountered during a pure testing process is termed as the observed reliability. If none of the errors are shared by more than one test case, the observed reliability is equal to the intrinsic reliability. The intrinsic reliability is always equal or greater than the observed reliability. We name this phenomenon as shielding effect since the perfect correction of a detected error on a tested flowpath shields our further observations of the same error out of other tests. The shielding

¹ Since an error may not cause all the six tests that lead to the faulty node, node 5, fail, the actual failure number may be less than six, with six as the maximum number of failures caused by node 5.

effect is measured by the ratio of the observed reliability over the intrinsic reliability, R_o / R_i , which is always bigger than or equal to one.

There is one thing that we need to be very careful about, that is, neither the intrinsic reliability nor the observed reliability estimated from the number of errors encountered during testing of the tested flowpaths is the correct reliability estimation for the tested flowpaths. The reason is that the fact we use to estimate the reliability level of the tested flowpaths is the number of failure-free tests performed on a number of tested flowpaths. Since as mentioned in previous section of this thesis, it is required that we remove all the detected error without introducing new errors. From testing data obtained from the tested part of the target software, we can estimate both the intrinsic reliability and observed reliability of the tested flowpaths. But the reliability estimated for the tested flowpaths is always higher than the intrinsic reliability and observed reliability of the same part of the same software.

Then what is the incentive to estimate the intrinsic reliability and the observed reliability of the tested flowpaths, assuming no correction has been made? Suppose we will not be able to test the untested flowpaths, we would like to estimate its reliability from the testing data obtained for the tested flowpaths. As mentioned above, there are lots of similarities between these two parts. Thus the error data from the tested portion must be able to give us some hints about the untested portion. By fully separating the two parts, the tested and untested, while estimating their reliability levels, we will under-estimate the reliability of the untested. This will lead to conservative reliability estimation, which is, for our purposes; good enough because we are dealing with software with high safety requirement and the true reliability is always higher or equal to our reliability estimation value. What we ignore when separating the two parts is the benefit brought about to the untested flowpaths by correcting the errors detected on the tested flowpaths. Thus, if any of the untested flowpath shares the same error with one of the tested flowpaths, the reliability we estimate for the untested flowpaths is smaller, which makes our reliability estimation conservative.

Now let us estimate reliability of the untested flowpaths assuming they have no sharing faulty node with the tested flowpaths that has been corrected. If we were to stop test at this point and put the software under real operating environment, what is the reliability level we expect to see from the untested flowpaths of the software? This process is exactly the same as we test the untested flowpaths without making any correction to it. The reliability we anticipate to see is the observed reliability of the untested flowpaths. Without worrying about the correlation between the two parts, are we going to see the same observed reliability as from the untested flowpaths? If this is true, we can use the observed reliability estimated from the number of errors detected and corrected during testing and perfect correcting process of the tested flowpaths as an estimate of the reliability we will see on the untested flowpaths in reality. But unfortunately this is not true. Though, it is easy to see that the intrinsic reliability level of the two parts of the same software is the same, how many times we observe a failure due to an error depends on the complexity of the underlying software. The more complex a piece of software is, the more observations we get from one software error. This makes perfect sense since the more complex a piece of software is, the more flowpaths, or test data are probable to share the same error.

Keep this in mind; we treat the untested flowpaths and tested flowpaths as two separate pieces of software with the same intrinsic reliability but different observed reliability depending on their complexity. Under this model, our question becomes simpler and clearer. The unknown is the observed reliability of the untested flowpaths. The knowns are the intrinsic reliability of the untested flowpaths and the intrinsic as well as the observed reliability of the tested flowpaths. The relation between intrinsic and observed reliability of the same piece of software is solely decided by its complexity. In this thesis, we use size, or the number of flowpaths of the software to represent its complexity. We should establish a relation between the intrinsic reliability and observed reliability as a function of the number of flowpaths in the software. For example, if a

linear relation is used. Given we have 10 failures out of 5 errors from the tested flowpaths, we would expect to see 5 failures out of 5 errors from the untested flowpaths if the size, or number of flowpaths in the tested group is twice the size, or number of flowpaths in the untested flowpaths.

Here, we propose a model to describe the observed reliability level for the untested flowpaths as a function of the number of flowpaths tested. The model is described in Figure 3-5. The vertical axis is the ratio between observed reliability and the intrinsic reliability of the untested flowpaths. This ratio is always bigger than one and smaller than some maximum value, it is a monotonously decreasing function of the size of the tested flowpath group. The horizontal axis is the number of flowpaths tested. The more the software is tested, the bigger the number of the tested flowpaths, the less complex the untested flowpaths, the smaller the ratio of observed reliability over the intrinsic reliability of the untested flowpaths. As the more commonly shared faulty nodes are more easily to reach, they tend to be gotten rid of at early testing and modifying stages. As a result, the curve decreases dramatically as the number of tested flowpaths increases. To start, we suggest a function format between $R_o / R_i(\text{untested})$ and N_T as:

$$R_o / R_i = 1 + aN_T^{-b} \quad \text{Eq. 3-65}$$

R_o	Observed reliability of the untested flowpaths
R_i	Intrinsic reliability of the tested flowpaths
N_T	Number of tested flowpaths
a	Function parameter, indicating the beginning R_o / R_i
b	Function parameter, indicating the decreasing speed of R_o / R_i as number of tested flowpaths increases

Figure 3-5 Ratio of Observed Reliability over Intrinsic Reliability for Untested Flowpaths

The function format should be proper for any types of software. Experiments on the software under consideration or similar programs should be performed to get approximation for

a and b . $R_o / R_i(\text{tested}) = f(N_T)$ has the same format as $R_o / R_i(\text{untested}) = f(N_U)$ since both of From

$$R_o / R_i(\text{untested}) = 1 + aN_T^{-b} \quad \text{Eq. 3-66}$$

we get

$$R_o / R_i(\text{untested}) = 1 + a(N - N_U)^{-b} = f(N_U) \quad \text{Eq. 3-67}$$

which then means

$$R_o / R_i(\text{tested}) = 1 + a(N - N_T)^{-b} \quad \text{Eq. 3-68}$$

The basic idea of the experiment is to keep records of the ratio $R_o / R_i(\text{tested})$ of the tested flowpaths as well as the number of tested flowpaths N_T at several points during testing process. Parameters of a and b are estimated from $R_o / R_i(\text{tested}) = 1 + a(N - N_T)^{-b}$

Eq. 3-68. In order to do the experiment, two copies of the target software are called for. The same testing input data are applied to both copies. If a failure is encountered, we correct the error immediately in one copy while leave the other copy un-touched. The same procedure is maintained as testing going on. Due to the shielding effect, we expect to see more failures in the untouched copy than the perfect correcting upon failure copy. Notice that the reliability estimated from the error numbers observed from the immediately perfect error removal version is not the reliability of the tested flowpaths, since the fact we use to update reliability of the tested flowpaths is always in the form of a number of failure free test cases due to the perfect error removal requirement.

Now, let us discuss how to estimate observed reliability and intrinsic reliability of the tested flowpaths. The methods for both reliability estimations are exactly the same. The facts utilized to make the approximations are of the same form as some number of failures encountered upon some tests performed. Though, out of the same group of test cases, the number of failures encountered in the perfect error removal case, which leads to the intrinsic reliability estimation, is less than that encountered in the no-touched case, which leads to the observed reliability estimation. From now, let us use unreliability instead of reliability as we estimate reliability of the tested flowpaths. The overall reliability under any cases is just one minus the corresponding unreliability:

$$R = 1 - \theta \quad \text{Eq. 3-69}$$

Now let us solve the unreliability estimation problem. It is the same for the observed reliability and the intrinsic reliability. The fact form we use to update unreliability for both case is the same, i.e., we find N_E faulty flowpaths out of N_T tested flowpaths. From the same set of input data, N_E from the untouched version is always greater than N_E from the perfect modified version. Again the overall unreliability is the weighted average of unreliabilities of all the tested flowpaths:

$$\theta = \frac{\sum_{j=1}^{N_E} p_{visit,j} \hat{\theta}_j + \sum_{k=1}^{N_T-N_E} p_{visit,k} \hat{\theta}_k}{\sum_{j=1}^{N_E} p_{visit,j} + \sum_{k=1}^{N_T-N_E} p_{visit,k}} \quad \text{Eq. 3-70}$$

θ	Observed / intrinsic unreliability of the tested flowpaths
$p_{visit,j}$	Visiting frequency of faulty flowpath j during testing
$\hat{\theta}_j$	Estimated unreliability of faulty flowpath j
N_E	Number of faulty flowpaths during testing
$p_{visit,k}$	Visiting frequency of failure free flowpath k
$\hat{\theta}_k$	Estimated unreliability of failure free flowpath k
N_T	Number of tested flowpaths

The visiting frequencies $p_{visit,j}$ and $p_{visit,k}$ can be calculated as:

$$p_{visit,j} = \frac{t_{visit,j}}{\sum_{j'=1}^{N_E} t_{visit,j'} + \sum_{k'=1}^{N_T-N_E} t_{visit,k'}} \quad \text{and} \quad p_{visit,k} = \frac{t_{visit,k}}{\sum_{j'=1}^{N_E} t_{visit,j'} + \sum_{k'=1}^{N_T-N_E} t_{visit,k'}} \quad \text{Eq. 3-71}$$

$t_{visit,j} / t_{visit,k}$ Number of visits to flowpath j or k

For a failure free flowpath, its unreliability $\hat{\theta}_k$ can be updated as any one flowpath described in the previous section, “Estimate the Average Unreliability of the Tested Flowpaths”.

Now let discuss how to estimate unreliability of a faulty flowpath, $\hat{\theta}_j$. The same as an failure free flowpath, $\hat{\theta}_j$ can be estimated as:

$$\hat{\theta}_j = p_{e,j} \times p_{f,j} \quad \text{Eq. 3-72}$$

$p_{e,j}$	The probability that there is an error on flowpath j
$p_{f,j}$	The probability that the error on flowpath j is encountered during next execution on this flowpath

If the error we found on flowpath j is of type I,

$$p_{e,j} = p_{f,j} = 1 \quad \text{Eq. 3-73}$$

The reason is that if we know there is a type I error on the flowpath, the probability there is an error on the flowpath is 1, unity ($p_{e,j} = 1$) and according to the definition of type I error, it causes the software fail every time it is visited, i.e. $p_{f,j} = 1$. Insert $p_{e,j} = p_{f,j} = 1$

Eq. 3-73 into $\hat{\theta}_j = p_{e,j} \times p_{f,j}$ **Eq. 3-72**, the unreliability of every faulty flowpath that has a type I error has an unreliability value of 1, i.e.

$$\hat{\theta}_j = 1 \quad \text{Eq. 3-74}$$

given there is a type I error on flowpath j .
If the error we found on flowpath j is of type II,

$$p_{e,j} = 1 \quad \text{Eq. 3-75}$$

We know for certain that there is an error on the flowpath. Given the definition of type II error,

$$p_{f,j} = p_{f,II} \quad \text{Eq. 3-76}$$

Insert $p_{f,j} = p_{f,II}$ **Eq. 3-76** into $\hat{\theta}_j = p_{e,j} \times p_{f,j}$ **Eq. 3-72**, if we find a type II error on flowpath j , its unreliability is estimated as:

$$\hat{\theta}_j = 1 - p_{f,II} \quad \text{Eq. 3-77}$$

Now, let summarize the reliability estimation process for the untested flowpaths. As testing proceeds, two versions of the target programs are kept. The same testing data are applied to both versions. As any failure occurs, we perfectly modify one version and leave the other version untouched. From both versions, the encountered errors are recorded. We stops the tests at a number of points, a series of the observed reliability $R_o = 1 - \theta_o$ from the untouched version, intrinsic reliability $R_i = 1 - \theta_i$ from the immediately modified version are calculated according to

$$\theta = \frac{\sum_{j=1}^{N_E} p_{visit,j} \hat{\theta}_j + \sum_{k=1}^{N_T - N_E} p_{visit,k} \hat{\theta}_k}{\sum_{j=1}^{N_E} p_{visit,j} + \sum_{k=1}^{N_T - N_E} p_{visit,k}}$$

Eq. 3-70 are estimated and their corresponding numbers of tested flowpaths N_T are recorded. After a number of data point is $R_o / R_i(\text{tested}) = 1 + a(N - N_T)^{-b}$ **Eq. 3-68** are collected, parameter a and b are estimated accordingly. $R_o / R_i(\text{untested})$, hence, can be calculated from $R_o / R_i(\text{untested}) = 1 + aN_T^{-b}$

Eq. 3-66. The last $R_i(\text{tested})$ is used as the intrinsic reliability of the untested flowpath, assuming the intrinsic reliabilities for both tested and untested flowpaths are about the same.

Finally, the observed reliability of the untested flowpaths, $R_o(\text{untested})$, which is the value to be used as the reliability estimation of the untested part of the target software is simply,

$$R_o = [R_o / R_i(\text{untested}) \times R_i(\text{untested})] = [R_o / R_i(\text{untested}) \times R_i(\text{tested})] \quad \text{Eq. 3-78}$$

There is one last comment on estimating unreliability of the untested flowpaths. As can be observed easily from Figure 3-4, when the number of tested flowpaths increases R_o / R_i decreases dramatically with an asymptotic value of 1. This is because those errors that affect more flowpaths tend to be detected during earlier testing stage, which are also removed quite early. After some tests, the errors left are those that have less impact, most of which are input data specific. If only a single input data set can trigger each error left in the untested flowpaths, $R_o / R_i = 1$, i.e., the intrinsic reliability is equal to the observed reliability that we care about. Then we can use the intrinsic reliability of the tested flowpath to approximate the observed reliability of the untested flowpaths.

3.3.3.2.5. Estimate the Number of Possible Flowpaths, $p_{\text{visit},T}$ and $p_{\text{visit},U}$

We have discussed how to estimated reliability of the tested flowpaths and that of the untested flowpaths. But we should not forget that one very important presumption, that is, the total number of executable flowpaths, N , is known. Without N , we do not know how much percentage of the program has been tested. It is obvious that the tested and modified flowpaths are more reliable than the untested ones; we can get reliability estimations for both parts. But how important of the relatively unreliable untested flowpaths regarding to visiting frequency compared to the tested flowpaths? To put it in another way, what is the odd that one of the untested flowpaths will be executed upon next execution, $p_{\text{visit},U}$? Apparently, the probability that a tested flowpaths will be executed next time, $p_{\text{visit},T} = 1 - p_{\text{visit},U}$. Until now, we have not mentioned anything about this topic. Will this question be answered if we can estimate the total number of flowpaths?

In this section, two experimental approaches to address these problems are presented, before which the unfeasibility of a theoretical solution is stated.

3.3.3.2.5.1. Unfeasibility of Theoretical Solution

Recall the definitions of the control flowgraph of a program and flowpath. The control flowgraph is simply a tree made up of nodes and links and a flowpath a sequence of nodes and links on the flowgraph. It is the decision nodes that decide how many possible flowpaths there are in a program. Starting from a decision node, the number of its direct successor nodes is the number of ways the program can go from that point. Theoretically, if we traverse from the bottom nodes on the flowgraph and keep going backward, we can calculate the number of possible flowpaths in a program.

Figure 3-6. Calculate Number of Flowpaths Theoretically

Take the flowgraph in Figure 3-6 an example. The theoretical number of flowpaths in a program can be calculated using depth-first traversal method from its flowgraph. In the example, there are 14 nodes, with node 1 as the top node, or starting node in the program and node 4, 9, 10, 12, 13, 14, 8 as possible exit nodes. According to the depth-first traversal method, suppose we have just reached node v , and let $w_1, w_2, \dots, w_k$ be its successor nodes from left to right. We shall next visit w_1 and then $w_2, \dots, w_k$ and go back to visit node v . Here, to calculate how many flowpaths there are in the flowgraph, we start from the top node, node 1. For every node, there is a recorder, storing the number of sub-flowpaths that lead to this node. In the beginning, we are at node 1. We first read the number saved in its first left successor node's, node 2's, recorder. Since we have not calculated the sub-flowpaths for node 2, there is no number saved there. Thus, from node 2, we go to node 4, the first left successor node of node 2. There is nothing saved there either. But for every bottom node, we assign 1 to its recorder. As every bottom node is an exit node of the program, there is only one way to going forward from there, i.e., exit the program. We put 1 into node 4's recorder and go back to node 2. From node 2, we visit its second left successor node, node 5. There is nothing recorded there. We go one step further and reach node 9. 1 is put into both node 9 and node 10's recorder since they are exit nodes. From node 10, we go back to node 5. Now all successor node of it have some number recorded. The total number of sub-flowpaths of node 5 is the total number of sub-flowpaths of its successor nodes, node 9 and node 10. Thus we save 1 plus 1, 2, into node 5's recorder. This process is repeated until the top node, node 1 is reached again with the total number of flowpaths in the program recorded in node 1's recorder. Refer to Table 3-1 to see the whole traversal and calculation process for the example flowgraph. There are 10 possible flowpaths in this program.

Node index, in order of sub-flowpaths calculation	Number of sub-flowpaths
4	1 (bottom node)
10	1 (bottom node)

11	1 (bottom node)
5	2 = 1 (node10) + 1 (node11)
12	1 (bottom node)
13	1 (bottom node)
14	1 (bottom node)
9	3 = 1 (node12) + 1 (node13) + 1 (node14)
6	3 = 3 (node9)
2	6 = 1 (node4) + 2 (node5) + 3 (node6)
7	3 = 3 (node9)
8	1 (bottom node)
3	4 = 3 (node7) + 1 (node8)
1	10 = 6 (node2) + 4 (node3)

Table 3-1 Calculate Number of Flowpaths of Program in Figure 3-6

As shown in the above example, if we do not take into account the relationship among input data, theoretically, the total number of flowpaths of a program can be calculated from depth-first traversal algorithm for graph structures. But, very often, due to special relationship among different input variables, some theoretically existing flowpaths may not be reached in practice. This already has been mentioned when we talk about white box testing strategy. A simple example would be two adjacent if/else selection structures whose decision making conditions include replicate variables. In such a case, some input data that lead to the if-block of the first if/else structure are possible to not able to reach either the if- or the else-block of the second if/else structure. Again, let us take the flowgraph in Figure 3-6 as an example. If the input data that lead the program to node 2 can only lead the program to node 12, neither 13 nor 14 while the input data that lead the program to node 3 can only lead the program to node 13 and 14, not 12, then only all the 10 possible flowpaths calculated theoretically can be reached in reality. As a result, assuming all the other flowpaths are reachable, there are only 7 possible flowpaths. Flowpath 1-2-6-11-13, 1-2-6-11-14 and 1-3-7-11-12 cannot be reached. In reality, there are lots of similar situations that make the theoretically calculated total number of flowpaths almost useless. Furthermore, to perform a depth-first traversal through out a real software control flowgraph is not anything close to an easy task, considering all the looping structures, multiple-use function blocks, even though the process is very simple and clear in the example control flowgraph shown in Figure 3-6.

Now, let us look at the probability that an untested flowpath is going to be executed next time, $p_{visit,U}$. Notice that $p_{visit,U} \neq \frac{N - N_T}{N}$ because different flowpaths have different probability to be accessed during operation. It is almost always true that after a lot of tests have been done on a program, the tested flowpaths always have better chance to be visited next time if the test data sampling process follows the same underneath distribution. This means $p_{visit,U} \ll \frac{N - N_T}{N}$. But how small is number is? For the total number of flowpaths in a program, at least we can get some hints from its structure while for the visiting probability, nothing can be obtained through any theoretical method.

Disappointed by the theoretical means of estimating N , $p_{visit,T}$ and $p_{visit,U}$, it is very natural for us to turn to experimental approaches as the theoretical method is neither capable of providing with good answers nor feasible to achieve. In the followed two sections, two

experimental methods to estimate total number of possible flowpaths in a program, as well as $p_{visit,T}$ and $p_{visit,U}$ are discussed.

3.3.3.2.5.2. First Experimental Solution

In this and the following sections, two empirical solutions to the total number of flowpaths problem are provided. They both appear naturally as the demonstration work, discussed in the next chapter, proceed, generating a huge number of data points. Though they are derived from observation of the pattern obtained from experimental data, there are very intuitive reasons supporting them. Both methods are black box method, meaning, in order to get the solution, we do not need to examine the inner structure, or the control flowgraph of the software. But as a result, a relatively large number of testing data points is needed to give a good solution. Remember this process is exactly the same as the ant-in-maze problem. One input data set is like an ant. Every time, we let an ant go through the maze, with colored paint dropped along the path by the ant, one path is identified. If there are many possible paths from the entry to the exit of the maze, even a lot more ants are needed taking into account the fact that several ants may end of passing the same path. Furthermore, the solution will vary if the input data distribution changes to some extend, depending on the size of the input data. But as long as we are close enough to the operating profile, the estimation result will be proper for the final reliability estimation task since the hard-to-reach flowpaths during testing are also the flowpaths of less importance under real operating situation.

Figure 3-7 Ants and Maze

Figure 3-8 Identify Possible Flowpaths

We have seen the saturation effect in an early section, discussing complete white box testing. According to the input data distribution, some flowpaths are hit very often while some are rarely touched though they are possible to be reached. As more and more tests are performed on a piece of software, it gets more and more difficult to visit an untested flowpath. We call this phenomenon saturation effect since although lots of new input data are applied, the speed of identifying new flowpath is very slow. We can plot a diagram with the horizontal axis representing the cumulative number of unique tests performed², the vertical axis representing the cumulative number of unique flowpath identified through testing. While tests continue, the curve grows longer and longer with its slope flatter and flatter. Imagine at one point, we have tested all the reachable flowpaths, the slope becomes zero and stays zero after that point since whatever input data we apply after that point, no new flowpaths will be identified, which means the curve do not grow vertically. But in practice, it is almost impossible to reach this point. An easy solution is to use an analytical curve, e.g. a polynomial curve, to fit the experimental diagram and extend the analytical curve to the saturation point, that is, the point where the slope reaches zero.

² If at least one input variable of the current input data set has a different value from any of the other input data sets, the test caused by this input data set is called a unique input data set.

Figure 3-9 Estimate Total Number of Flowpaths, Experimental Approach1

In Figure 3-9, the program has been tested x_0 times³, out of which, N_T unique flowpaths are identified. The irregular curve is the experimental curve. The regular curve is the fitting curve obtained from the experimental one. Based the shape of the experimental curve, several function form can be used for the analytical fitting curve. A simple example would be a second order polynomial function, $y = f(x) = a + bx + cx^2$. Parameters, a , b and c can be estimated from experiment data. We call the point where the slope of the fitting curve becomes zero as the saturation point, after which, no new flowpath will be identified since every flowpath has been tested at least once. If the function chosen only has one inflection point, that point is the saturation point. If we choose $y = f(x) = a + bx + cx^2$, the total number of tests needed to cover all possible flowpaths, or the x coordinate, x^* , of the saturation point can be obtained by:

$$\frac{df}{dx}(x = x^*) = b + 2cx^* = 0 \quad \text{Eq. 3-79}$$

We get:

$$x^* = -\frac{b}{2c} \quad \text{Eq. 3-80}$$

The total number of flowpaths can be estimated as:

³ x_0 tests are not necessary to be unique.

$$\hat{N} = y^* = f(x^*) = a + b\left(-\frac{b}{2c}\right) + c\left(-\frac{b}{2c}\right)^2 = a - \frac{b^2}{2c} + \frac{b^2}{4c} = a - \frac{b^2}{4c} \quad \text{Eq. 3-81}$$

As we take the fitting curve as an approximation of the real experimental curve, the probability of visiting a tested flowpath next time can be estimated as:

$$p_{visit,T} = \frac{x_0}{x^*} \quad \text{Eq. 3-82}$$

and $p_{visit,U} = 1 - p_{visit,T}$.

Up to this point, we have described the first experimental method to estimate the three important parameters, N , the total number of flowpaths can be reached, $p_{visit,T}$, the probability that a tested flowpath will be executed upon next operation and $p_{visit,U}$, the probability that an untested flowpath is going to be visited. They are essential in estimating the reliability of a program based on its structure. It is a very simple approach with limited and simple calculation involved. The experimental data required are the number of tests made and the number of unique flowpath identified through the test data. There is one concern about this method. For some cases, when the last testing point is very close to the saturation point, the estimated total number of flowpaths may be smaller than the total number of flowpaths identified, which is certainly wrong. This happens since the difference between N and N_T is so small, the error of the fitting curve from the experimental curve is much bigger than this difference. If this happens, this approach cannot be used to estimate N , in which case, the second experimental approach has to be utilized. But it is still useful in estimating $p_{visit,T}$ and $p_{visit,U}$.

Figure 3-10 Predicted Total Flowpath Number $\hat{N}$ Smaller than Tested Flowpath Number N_T

3.3.3.2.5.3. Second Experimental Solution

The first experimental solution to N , $p_{visit,T}$ and $p_{visit,U}$ is definitely the most straightforward answer. But if the $\hat{N} < N_T$ problem occurs, it becomes useless in estimating N , the total number of flowpaths, although, from the result, we can say N is very close to N_T and we have almost tested all the possible flowpaths. But neither “close” nor “almost” is a good or scientific answer. As we have been talking about probability throughout this thesis, we prefer an answer with probability value associated with it. We should be able to answer how close we are from the complete flowpath coverage testing point with what probability. Because N is not a precise number, there should be a range around it with a likelihood value. In this section, we will try to address these questions by using the almost the same group of data points as in the first experimental approach. Though the reasons behind this approach are not as obvious, they are still as solid as the first approach.

Again, a presumption to apply this method is that the input data are sampled randomly according to certain distribution, in most cases, preferably, the operating profile. Some flowpaths are more easily to be visited and some are not. The ones that are easily reached will be identified earlier. Let us look at the same phenomenon from another perspective. If the underlying probability distribution of the input data does not change over time during testing, the earlier identified flowpaths are the ones that have more visits on them. We can sort the flowpaths according to the order they are identified. After many random tests, the number of visits to each flowpaths can be plotted. We can imagine that there is a trend going through all the data points.

Figure 3-11 Ideal Number of Visits to Flowpaths According to Identification Order

In

Figure 3-11, an extreme situation is shown. The horizontal axis is the index of identified flowpaths lined up according to their order of recognition time. The vertical axis is the number of visits of each flowpath. The flowpaths with smaller indices always have better chances of being hit during testing. If the input data follow the same distribution as in actual situation, the same flowpath visiting time distribution would be the same as can be seen during testing. In this extreme example, if the curve follows a perfect analytical function, the point where the curve crosses the horizontal axis tells us the total number of flowpaths in the software. It is also possible that the curve does not interconnect with the horizontal axis, e.g. it has an exponential form. In this case, if we can connect this experimental curve with visiting probability density distribution function of the flowpaths, some very useful results can be obtained.

For now, let us assume we know the analytical form, $y = f(x)$, of the curve shown in Figure 3-11, where x is the flowpath index and y is the number of visits during testing. If we have performed a large number of tests such that the curve is statistically significant, the number of visit to each flowpath y is proportional to the probability it is hit by an execution of the program. The only difference between $f(x)$ and the probability density function $P(x)$ is a normalization constant. Thus we have:

$$P(x) = \frac{f(x)}{\int_0^{\infty} f(x) dx} \quad \text{Eq. 3-83}$$

Figure 3-12 Estimate Untested Flowpath Visiting Probability $p_{visit,U}$

If we have tested N_T flowpaths, the probability that a non-tested flowpath will be visited upon next execution can be estimated as:

$$p_{visit,U} = \int_{N_T}^{\infty} P(x)dx \quad \text{Eq. 3-84}$$

All the discussions in this section until now are based on the assumption that we know such an ideal analytical function connecting the flowpath index and its number of visits during testing. But unfortunately, in reality, this is never true. Now, we need to approximate such a function form from our experimental data. In real life, though the trend that earlier identified flowpaths tend to have more visits during testing, there are lots of ups and downs among contiguous flowpaths regarding their number of visits. To put it another way, there are lots of noises if we directly connect all the experimental data points. Our last task in this section is to get rid of the noise and abstract a curve with a known analytical form. If this is achieved,

$$p_{visit,U} = \int_{N_T}^{\infty} P(x)dx$$

Eq. 3-84 can be used to make the estimation for $p_{visit,U}$.

There are two steps to jump from the noisy experimental curve to the desired smooth function. First, we can divide the tested flowpaths into contiguous groups. If each group contains k flowpaths, we have $n = \frac{N_T - \text{mod}(N_T / k)}{k} + 1$ groups. Within each group, the numbers of visits to all the flowpaths should be relatively close to each other since they are identified around the same time scale. For each group i , we calculate the average number of visits to each flowpath as:

$$\bar{y}_i = \begin{cases} \frac{1}{k} \sum_{j=1}^k y_j & i < n \\ \frac{1}{N_T - k \times (n-1)} \sum_{j=1}^{N_T - k \times (n-1)} y_j & i = n \end{cases} \quad \text{Eq. 3-85}$$

Now we can plot a new diagram $(\bar{x}_i, \bar{y}_i)$, where, for each group i , $\bar{x}_i$ is the middle value:

$$\bar{x}_i = \frac{x_{i \times k + 1} + x_{(i+1) \times k}}{2} \quad \text{Eq. 3-86}$$

The new plot has only $n \approx N_T / k$ data point with less noise. Proper k should be chosen according to the real experimental condition. If k is too big, the data points n will be too small. As a result, the resulting curve will not have enough statistical significance. If k is too small, each group containing only a very small number of data points to average, the noise-reducing effect will be very small.

We have talked about the first step. By taking average, we get a much smoother curve. But, still, it is an experimental curve. In order to reach our purpose, an analytical form is needed. Again, the second step is to use curve fitting. First a suitable function form is chosen and then the parameters involved are estimated using the data points obtained through step one. For the function form, a simple example is the exponential function:

$$y = ae^{-bx} \quad \text{Eq. 3-87}$$

b can be estimated from $(\bar{x}_i, \bar{y}_i)$. a is obtained through normalization as:

$$\begin{aligned} \int_0^\infty ae^{-bx} dx &= 1 \\ \Rightarrow a &= b \end{aligned} \quad \text{Eq. 3-88}$$

Then the probability that an untested flowpath will be visited upon next execution is:

$$p_{\text{visit}, U} = \int_{N_T}^\infty be^{-bx} dx = e^{-bN_T} \quad \text{Eq. 3-89}$$

4. Demonstration

4.1. Introduction

This chapter presents the results of a case study in which the designed testing and reliability estimation strategy described in chapter **Error! Reference source not found.** is applied to a program used in a nuclear power plant's reactor coolant system. The program is the generic Signal Validation Algorithm (SVA) used for calculating and validating the values of parameters of the reactor coolant system. The 001 CASE tool is used to facilitate and automate the methodologies. Due to this reason, SVA is written in 001 CASE tool language. It needs to be noticed that 001 CASE tool is not the only tool available for the desired approaches. The work described in this thesis is a continuous work of an earlier Ph.D. thesis [**Error! Reference source not found.**], where all the details about why 001 is chosen were presented. The experiment reported here is aimed at investigating the feasibility of the proposed methods. The same as in any industry related project, it is vital to show that the intended academic work is able to be automated in the real world.

4.1.1. General

In this experiment, the original specification of SVA is developed into C code under 001 developing environment following DBTF described in chapter **Error! Reference source not found.** While developing, the variable and functional consistency and completeness are checked. The program then is tested against previously developed and tested software, obtained from our industry collaborator. Testing input data are sampled according to reasonable probability distributions obtained by examining nature of the program. If there is any failure detected, the program developed under 001 is debugged and retesting is performed for all the used input data to make sure no new error has introduced. While testing, the nodes and flowpaths of SVA is identified and recorded. Both the testing results and the identified flowpaths are used to estimate the unreliability of the software, θ , with both nodal coverage reliability estimation and refined feasible flowpath coverage reliability estimation techniques. The detected errors, identified flowpaths are analyzed in the end.

4.1.2. Working Path Flow

In this section, a working path flow of the demonstration work is presented. Details of each working step will be discussed more thoroughly later in this chapter.

Figure 4-1 SVA Demonstration Working Path

4.2. Sample Software — Signal Validation Algorithm (SVA)

In the nuclear power industry, many efforts have been developed to improve plant performance and availability. An important factor for the success of these efforts is the improved access to reliable information for the entire plant, especially for safety critical systems. Many nuclear utilities are installing or upgrading their plant computer systems to provide such a capacity to handle plant data and provide operators with data interpretations that are accurate, extensive, reliable and useful. The application of these computer systems provides operators with a ready access to a variety of sensor signals. It permits operators to use an increased number of redundant, independent measurements in the decision-making process during normal and emergency plant operations. However, simply displaying all these redundant measurements is not desirable because it may adversely complicate the operator's mental processing tasks of data evaluation and analysis. One way to reduce the operators' work load in comparing, analyzing, and evaluating sensor signals is to apply automatic signal validation algorithms as a pre-filter to reduce the amount of information passed to the operator and to the other systems in the plant.

The Signal Validation Algorithm (SVA) of this case study is a generic algorithm that performs signal validation for the plant processes that contain multiple sensors measuring the same or closely related process parameters. The algorithm is called generic because it is generally applicable to all types of parameters. Only slight modifications need to be made for each particular parameter according to different quantities of the sensor used.

4.2.1. SVA — the Algorithm

SVA is a generic validation algorithm to reduce unnecessary information loading. The algorithm takes the outputs of all sensors measuring the same parameter and generates a single output representative of that parameter, called the "Process Representation". A generic validation approach is used to ensure that it is well understood by operators. This avoids an operator questioning the origin of each valid parameter.

SVA averages all sensors and deviation checks all sensors against the average. If the deviation checks are satisfactory, the average is used as the “Process Representation” and it output as a “valid” signal. If any sensors do not successfully pass the deviation check against the average, the sensor with the greatest deviation from the average is taken out and the average is recalculated with the remaining sensors. When all sensors used to generate the average are deviation-check-satisfactory against the average, the average is used as the “valid” “Process Representation”. This “valid” “Process Representation” is then deviation checked against the Post Monitoring System sensors. If the second deviation check is satisfactory, the “Process Representation” is displayed with the message “Valid PAMI” (Post-Accident Monitoring Indication), indicating that this signal is suitable for monitoring during emergency conditions, since it is in agreement with the value as determined by the PAMI sensors. As long as agreement exists, this indicator may then be utilized for post-accident monitoring rather than utilizing the dedicated PAMI indicator. This provides a Human Factor Engineering advantage of allowing the operator to use the indicator he normally uses for any day-to-day work and which he is most familiar with.

4.2.2. Original Design Document of SVA

There is one volume of original SVA design document upon which this demonstration is based. It is a pseudo-program structure in English text that provides definitions of terms, functions, description of the data calculations, and logical connections between functions and calculation. The text occupies 14 pages. This is the document we used as specification, from which the demonstration software is built under 001 CASE tool.

4.2.3. Oracle

Whenever a software testing task appears, there should be some mechanism, an oracle that is able to determine whether or not the results of test execution are correct. There is a large literature that addresses the question of how to automatically generating oracle from formal specification of the target software, tabulating values of the input variables. This is, by itself, a big topic in software engineering community. In our work, oracle generating has not been a focus. Thus, as in many software testing research and practices, we assume that we have the mechanism, an oracle, that determine if the output from the software under testing is correct.

For our demonstration work on SVA, we have obtained a previously developed and tested program, obtained from our industry collaborator. This is a “quick C” program of a little more than 3000 lines, implemented on IBM/PC. We term this piece of software as oracle SVA from now on. During testing, for any given input data, if any output from the software under consideration (we term this piece of software as 001 SVA from now on since it is developed under 001 CASE tool) is of any difference from the output of oracle SVA, we claim a failure case. If this happens, debugging, retesting are performed on 001 SVA until all the outputs corresponding to all the tested input data are correct.

4.2.4. A Testing Document of SVA

The last document we obtained from our industry partner is the test evaluation report for SVA. The purpose of this document is to provide a record of the evaluation of the generic SVA program used as the oracle in the work we described in this chapter. This evaluation was performed prior to algorithm implementation in the mockup to ensure that the validation algorithm in the mockup is based on a sound design.

In the document, the testing purpose, scope, background, algorithm definition, software implementation, etc. are covered briefly. Ten test cases were designed and their desired outputs

of the algorithm were identified prior to testing. Both the test cases and correct outputs were based on operational requirements during both normal and emergency conditions that require monitoring processes containing multiple “like” parameters. The snapshots of the testing screens, including both input screens and result screens, were shown in the report. For the failures occurred during testing, causes were analyzed.

According to the methods described in last chapter, this document is not essential for the purpose of demonstration, as opposed to the prior two documents, the software specification and the oracle program. Furthermore, the approach to pick software input data in the report is not applicable to the approaches we have described. As required by our thorough testing objective, automation in every step is crucial to the success of our work reported in this thesis. But still, this document is very useful to us. Compared to the specification of the software, this software evaluation document provides us with a more detailed picture about the types of input data that most normally showed up to SVA. In the specification, there are only descriptions about the input variables, their functions in the algorithm, etc. It is the testing document that gives us some numbered examples about how the input data look like. With little knowledge about the input domain, the input data probability distribution functions are the joint product of both the input variable definitions in the specification document and this testing report.

4.3. 001 CASE Tool

The 001 CASE Tool is an integrated system and software development environment. An automation of Development Before The Fact (DBTF) approach, it is used to define and generate itself.

4.3.1. DBTF

Already discussed in chapter **Error! Reference source not found.**, Development Before The Fact (DBTF) approach is the formal software development technique we have chosen for this study. It is a software development methodology marketed by Hamilton Technology Inc., of Cambridge. DBTF has been successfully applied in many large industrial projects and is being marketed commercially.

What makes DBTF different is it is a preventative paradigm instead of a curative one. Problems associated with traditional methods of design and development are prevented "before the fact" just by the way a system is defined. That is, DBTF concentrates on preventing problems of development from even happening; rather than letting them happen "after the fact", and fixing them after they've surfaced at the most inopportune and expensive point in time.

4.3.2. 001 System

001 (pronounced "double oh one") is a fully integrated systems engineering and software design and development environment. Being the application of DBTF methodology, 001's motivation is to facilitate the "doing things right in the first place" development style, avoiding the "fixing wrong things up" traditional approach. To automate the theory, 001 is developed with the following considerations: error prevention from the early stage of system definition, life cycle control of the system under development, and inherent reuse of highly reliable systems. It can be used to define, analyze and automatically generate complete, integrated, and fully production-ready code for any kind or size of software application with significantly lower error rate and significantly higher usability than traditional approaches.

4.4. Capture SVA Specification into 001 TMaps and FMaps

4.4.1. Formal Structures Building

In our experiment with SVA, we skip the process of requirement definition in the form of a user's English document. We use the same specification from out of which the oracle SVA program was built in order to compare outputs from both programs. Our first task is to transfer the specification of SVA into 001 TMaps and FMaps. Then FMaps and TMaps are analyzed, continuing with the automatic generation of production ready code in C. Before C code for each function or FMap is linked together, modular testing is performed, with special emphasis on key FMaps.

There are totally 21 TMaps defined for SVA program. They can be categorized into two major groups. TMaps in the first group are data types associated closely with SVA, e.g. sensor readings, criteria data, sensor status, etc. TMaps in the second group are data types defined for testing purpose, e.g. test case data structures, node and flowpath recorders, etc. Based on the 21 TMaps, FMaps are defined from SVA specification. There are totally 156 independent FMaps used to specify the main operation of the algorithm and the supporting modules.

4.4.2. Specification Incompleteness and Inconsistency Captured

During the formal specification capturing process within 001, 6 classes of problems in the original SVA specification document are found. Refer to table for details.

Class Number	Problem Description	Number of Errors in Class
1	Ambiguities in using words or terms	16
2	Inconsistencies in defining and using words or terms at different places in the document	11
3	Ambiguities in defining operations and functions	10
4	Inconsistencies in defining operations and functions at different places in the document	1
5	Incompleteness of the logic design of operations and functions, based on the data type structures of the input variables	6
6	Incompleteness of output state definition of operations and functions, based on the data type structures of the input variables	4

Table 4-1 Classes of Problems Raised in Mapping SVA Specification

4.4.3. Modular Testing

The testing of the captured specification is performed in two stages: a modular testing stage in which each of the modules is tested partially or completely based on its importance and an integrated testing phase in which the entire program is under testing. Though both of them are testing procedures, they are quite different due to the size of the system under consideration. As every module, or FMap in 001, is very small and simple, it is not very hard to exercise a complete flowpath testing if it is necessary. As for the whole software, it is much more complicated, instead of a feasible complete flowpath testing; some compromise is needed depending on the size of the program under question. In our experiment, we apply two very different strategies for each case. In this section, the modular testing, the easier one is discussed briefly and the integrated testing task is left until next section, which is covered in much more detail.

To perform modular testing, the Executor of 001 Tool Suite is frequently used to simulate behavior of a section of the whole program (a module or FMap) in order to capture all potential failures at an earlier stage.

For very important modules, feasible complete flowpaths testing can be performed. For detailed descriptions of complete flowpath coverage testing, refer to [Litt92]. Simply speaking, the process is as follows. First, the flowgraph of the given module is constructed. In 001, it is just a simplified FMap. Second, the looping structures within the flowgraph are expanded, to zero repetition, one repetition and two repetition. Third, all the flowpaths are identified on the expanded flowgraph. For each flowpath, in order to improve the error-revealing capability, multiple test cases are generated for each path whenever is possible. In the end, testing is performed on the module. The tester should decide whether or not a test is a success or a failure based on his knowledge about the functionality of the underlying module. As mentioned in [Litt92], the whole process has to be performed manually. Since every module is relatively small, having a small number of possible flowpaths, the above described process can be done on some key modules of the given software.

Besides the formal flowpath testing method, another loosely constructed test strategy could be performed as well. Based on the functionality of the module, from the specification, different operational scenarios of the module could be identified. For each scenario, several input data of the module could then be created to test the underlying FMap. The same as the above feasible complete flowpath testing technique, the correct output for each set of input data set is recognized manually.

Due to the natures of the above two modular testing methods, the programmer who has created the module has to be the examiner of the module. Refined knowledge about both the functions and the structures of the modules under testing is required for the modular testing purpose. Though this is against the widely accepted rule for testing, i.e. different groups of people should be involved in the development and test procedures, as an intermediate checking step, modular testing is still every useful in early detection of possible errors in the program.

4.5. Integrated Testing on SVA

Until now, each module has been extensively tested and is expected to have fairly low failure rate. All the modules have been linked with 001 and ready for integrated test. Although modular testing is very important, the integrated test is always indispensable. Since even all the modules are performing perfectly according to their specifications, interface errors may still exist, which can never be detected until the integrated testing is exercised. It is almost always impossible to follow the steps to perform a feasible complete flowpath coverage testing as practiced in modular testing. The reason is simply that the whole program is too complicated to be traced. Even though the flowgraph can be expanded, for most of the software system, the task of identifying input data corresponding to certain flowpaths is too intricate. If we do not have any automatic tools and have to do all the flowpath and input data identification all by hands, which unfortunately turns out to be true in our case, even all the constraints are combined linearly, the undertaking is never trivial. In order to reduce the task, as already discussed in chapter **Error! Reference source not found.**, we will reverse the step order. Instead of specifying a path and trying to find the input values that cause the path to be traversed, we specify some input values and try to identify the traversed flowpaths, which can be done automatically. The overall effect of using this approach is the same as identifying the paths first and chooses the input data later. This is exactly what we have discussed in chapter **Error! Reference source not found.** as grey box testing.

In the following subsections, how we have achieved the grey box testing on SVA is discussed in much more detail.

4.5.1. Testing Data Generation

We have applied three different testing data generating strategies for integrated testing upon SVA 001 program. They are manually designing input data from operational profile / software specification, randomly sampling input data based on functionality and software logic of SVA, manually identifying input data to cover un-visited software nodes. We use the three techniques accordingly, i.e., we manually pick some input values first to test the best known and mostly experienced scenarios; generate a huge amount of input data according to some distribution, trying to cover most of the nodes; finally some input data again need to be selected carefully to test those untested nodes. If we can tell from the testing results that even we have exercised a lot of tests on the program, the possibility of identifying untested flowpaths is un-ignorable; more random sampling of the input data can be practiced to approach a more complete feasible flowpath testing. Among them, there are two input data generating processes that are done by hands. As a consequence, they only contribute a tiny percentage of input data compared to the random sampling method. But still they are very important, especially the last one, without which, some tricky part of the software (those nodes that have not been covered after quite a large amount of test cases are performed) is very hard, if not at all, to be tested.

4.5.1.1. Input Identification from Operational Profile

As we do not have any direct information about the operational profile of SVA, we have to refer to some roundabout document. In this case, we choose the specification. In the specification, most of the major logics are described clearly. As the first barrier toward any failure involved with these most frequently occurring scenarios, test cases are especially designed to test these scenarios. All the input data in this category are picked manually. We developed about 300 test cases according to this technique. 2 failures happened, out of which we found 3 errors. For more details about the encountered failures and detected errors, see the testing results section 4.5.4.

Though none of the test cases created from the specification are tricky in the sense that none of them can help the tester to find unexpected errors if the program has done a perfect job in following the specification, still this traditional way of generating test cases are very important and should always be practiced ahead of all the other approaches. If the errors found out of the input data generated under this technique were left in the program, the impact would be much worse than the errors that can only be detected by much trickier input data, which are not as easy to be encountered during field use.

4.5.1.2. Random Sampling

Here comes the technique we use to generate the majority of test cases we apply to SVA 001 program. In total, we have generated 198,321 test cases, out of which, 133,093 are unique. Except the 300 or so input data sets picked up in line with the specification document and 84 test cases chosen specifically to cover the untested nodes after 38,717 tests, all the input data are generated in accordance with the method described in this section. Though the efficiency of detecting errors is not as high as the other two techniques, the method is essential in discovering the failures triggered by unanticipated operating conditions.

First, let us clarify what the SVA input variables are. SVA is a dual range application where values of two similar sets of variables, one for narrow range and one for wide range, need to be specified to perform one test. The input variables or parameters are as follows.

Narrow range parameters:

- Four similar but independent sensor readings; denoted by “s1”, “s2”, “s3” and “s4” respectively;

- Instrument uncertainty for the above sensors; denoted by “*iu*”;
- Expected process variation for the process containing the four sensors; denoted by “*ev*”;
- Narrow range, including narrow higher bound and lower bound; denoted by “*lb*” and “*hb*” respectively;

Wide range parameters:

- Two similar but independent PAMI sensor readings; denoted by “*p1*” and “*p2*”;
- Instrument uncertainty for the PAMI sensors; denoted by “*iu_p*”;
- Expected process variation for the process containing the two PAMI sensors; denoted by “*ev_p*”;
- Wide range, including wide higher bound and lower bound; denoted by “*lb_p*” and “*hb_p*” respectively;

Additionally, a variable exists for an operator selected sensor value, denoted by *op*, which may be selected during algorithm execution in the event a validation fault or PAMI fault occurs.

Tests are to be accomplished by specifying values for each of the above parameters. Upon execution, the SVA 001 program runs through the algorithm and presents the result, which will be compared with the output from the oracle program as the consequence of the same input data set. Looking at the surface of these parameters, all of them are independent of each other. If this is the truth, our tasks would be much more straightforward. We only need to work out the range and construct a reasonable distribution for every input variable independently. Unfortunately, this is not true. The input variables are real world physical parameters, which should make solid physical sense in terms of their relative magnitudes.

The input variables have been divided into two groups, the narrow range parameter group and the wide range parameter group. Variables within both groups have identical relationships within their respective group. We will first present the technique we generate input data for the first narrow range group and transfer to the second wide range group by identifying the difference.

We start the narrow range group by opting for a domain. This is done manually based on the specification. Since it seldom needs to be changed, the manual work does not form a big trouble for the tester. This domain will be the main play ground for the input data within the first group. Let us represent the lower bound of this domain as “*MIN*” and the upper bound as “*MAX*”. The range between “*MIN*” and “*MAX*” is symbolized as “*R*”; and $R = MAX - MIN$. First, a lower bound of the narrow range is sampled uniformly between *MIN* and $MIN + \frac{3}{4}R$, i.e.,

$$lbR = uniform(MIN, MIN + \frac{3}{4}R) \quad \text{Eq. 4-1}$$

And then an upper bound is generated between the lower bound as:

$$ubR = uniform(lbR, MAX) \quad \text{Eq. 4-2}$$

Notice that we do not use lb and ub since they are still not the actual narrow range we shall use during testing. The generation of the narrow range boundaries is connected with wide range boundaries and is covered a little later.

40×4 independent numbers are randomly sampled between lbR and ubR , i.e.,

$$si_k = \text{uniform}(lbR, ubR), i = 1, 2, 3, 4 \quad k = 1, 2, \dots, 40 \quad \text{Eq. 4-3}$$

These 40×4 numbers are treated as forty sets of input sensor readings. Each set includes 4 readings corresponding to the 4 narrow range sensors. In Eq. 4-3, si_k means the value of sensor i in the k th sensor reading set.

The sampling process for the two narrow range criteria variables iu and ev is very simple. Two domain boundary constants are chosen in advance, CRI_MIN and CRI_MAX . Both narrow range criteria variables are sampled randomly within the domain (CRI_MIN, CRI_MAX) , i.e.,

$$iu, ev = \text{uniform}(CRI_MIN, CRI_MAX) \quad \text{Eq. 4-4}$$

All the input value sampling process for the wide range parameters are the same as for the narrow range parameters. $MIN_P < MIN$ and $MAX_P > MAX$ are chosen. We can get lbR_P and ubR_P as:

$$\begin{aligned} lbR_P &= \text{uniform}(MIN_P, lbR) \\ ubR_P &= \text{uniform}(ubR, MAX_P) \end{aligned} \quad \text{Eq. 4-5}$$

40×2 random numbers are generated for the forty sets of wide range sensor readings. Each set includes two sensor reading to corresponding to the two PAMI sensors or wide range reading sensors.

$$pi_k = \text{uniform}(lbR_P, ubR_P), i = 1, 2 \quad k = 1, 2, \dots, 40 \quad \text{Eq. 4-6}$$

CRI_MIN_P and CRI_MAX_P are chosen so that the criteria parameters for the wide range can be sampled as,

$$iu_p, ev_p = \text{uniform}(CRI_MIN_P, CRI_MAX_P) \quad \text{Eq. 4-7}$$

Now, we have get forty sets of input data, ten numbers for each set, except the range parameters. Within each set, we have six sensor readings, four narrow range sensor reading, $s1, s2, s3, s4$ and two wide range sensor readings $p1, p2$; four criteria parameter values, iu, ev, iu_p, ev_p . The only inputs we have not created are the range boundaries, for which we have already obtained $lbR_P < lbR < ubR < ubR_P$. We shall use these four numbers to get the range

boundaries. Four range settings can be obtained out of one set of lbR_P, lbR, ubR, ubR_P depending on the relative arrangement of the wide range and narrow range. They are,

$$lb_p1 = lbR_P, lb1 = lbR, ub1 = ubR, ub_p1 = ubR_P \quad \text{Eq. 4-8}$$

This means the wide range is truly wider than the narrow range at both ends, that is, $lb_p1 < lb1 < ub1 < ub_p1$. The second arrangement is,

$$lb_p2 = lbR_P, lb2 = lbR, ub2 = ub_p2 = ubR \quad \text{Eq. 4-9}$$

In this setting, the upper boundaries for both wide and narrow range are the same while the lower boundaries are different, i.e., $lb_p2 < lb2 < ub2 = ub_p2$. We also have,

$$lb_p3 = lb3 = lbR, ub3 = ubR, ub_p3 = ubR_P$$

$$lb_p4 = lb4 = lbR, ub4 = ub_p4 = ubR$$

Eq. 4-10

Figure 4-2 Four arrangement settings for one SVA input data set

Figure 4-2 shows the four arrangement settings used during SVA testing. Every one of the above generated forty ten-random-number input data will be put into test against the four arrangement settings described.

Do we have all the input data for test until now? If we were to do an interactive testing, that is one person will behaves like an operator. When the SVA program prompts for the operator to choose a sensor reading as the representative, this person responses according to the instruction. In order to save time, as we will mention in the next section, we have excluded all the human computer communication. Except those few cases, we want to test if the program

performs the correct function under erroneous situations; we use a random number generator to choose a number from zero to six to simulate a human being behavior. Since we have six sensors (four normal or narrow range sensors and two PAMI or wide range sensors), if a number from one to six is chosen, our SVA program believes the sensor reading corresponding to that number has been chosen. If number zero is chosen, that means the operator does not want to choose and the SVA program will try further to find the representative signal value.

To sum up, there are 13 numbers generated for each test. They are four normal sensor readings $s1, s2, s3, s4$ (4); two PAMI sensor readings $p1, p2$ (2); two normal sensor criteria numbers iu, ev (2); two PAMI sensor criteria numbers iu_p, ev_p (2); two narrow range boundaries lb, ub (2); two wide range boundaries lb_p, ub_p (2) and one operator selecting sensor (1). Out of the above described input data generating process, we can get 40×4 input data sets.

In generating test cases for SVA 001 program, besides the above random sampling method, we have also used another approach, which is only a little different. We employ Gaussian distribution instead of uniform distribution for all the parameter values obtained through random sampling process. In reality, Gaussian distribution method is the first we adopt in our test data generating process. After some tests, we believe that more diversified input data is more effective in testing those untested flowpath and hence change all the underlying distributions from Gaussian to Uniform. See Table 4-2 for the parameter random sample methods we use in this study.

Parameters		Uniform Distribution Method	Gaussian Distribution Method
Given values		MIN, MAX, MIN_P, MAX_P CRI_MIN, CRI_MAX CRI_MIN_P, CRI_MAX_P	MIN, MAX, MIN_P, MAX_P iu, ev, iu_p, ev_p
Calculated values		$R = MAX - MIN$	$lbR = MIN$ $ubR = MAX$ $cen = uniform(MIN, MAX)$ $var = (ev + iu / 2) \times 2$ $lbR_P = MIN_P$ $ubR_P = MAX_P$ $cen_p = uniform(MIN_P, MAX_P)$ $var_p = ev_p + iu_p / 2 \times 2$
Random Sampled Values	lbR	$lbR = uniform(MIN, MIN + \frac{3}{4}R)$	GIVEN
	ubR	$ubR = uniform(lbR, MAX)$	GIVEN
	si	$si = uniform(lbR, ubR)$	$si = Gaussian(cen, var)$
	iu, ev	$iu, ev = uniform(CRI_MIN, CRI_MAX)$	GIVEN
	lbR_P	$lbR_P = uniform(MIN_P, lbR)$	GIVEN
	ubR_P	$ubR_P = uniform(ubR, MAX_P)$	GIVEN

	pi	$pi = \text{uniform}(lbR_P, ubR_P)$	$pi = \text{Gaussian}(cen_p, var_p)$
	iu_p, ev_p	$iu_p, ev_p = \text{uniform}(CRI_MIN_P, CRI_MAX_P)$	GIVEN
	op	$op = \text{uniform}(0,6)$	$op = \text{uniform}(0,6)$

Table 4-2 Random Sampling Input Data for SVA

Following the above random input data generating process, huge amount of input data sets can be created with very little consuming time. Roughly speaking, the time spent in this process can be ignored within the whole testing time scheme.

4.5.2. Automatic Testing Process

We have made it clear that we want to test our program as thorough as possible due its safety-critical feature. The repetition of a large number of trials requires testing automation, which includes automated test case generation, automated input data feeding for oracle and target software, automated oracle output and testing output recording as well as comparison and automated flowpath keeping and analysis, etc. Except the random input data sampling process discussed in 4.5.1.2, all of these automation strategies are presented in this section.

4.5.2.1. Input Data Reading

- Oracle Input Reading

The original SVA program in quick C used as our oracle software is a human-computer interactive program. The sensor readings, criteria parameter values, range boundaries, etc. are input from the algorithm testing screen by the tester. The initial and resulting sensor status, alert messages, resulting representative signal values, etc. are displayed on the same algorithm testing screen. The original interface can be seen in Appendix 5.3. This I/O process is extremely time-consuming, preventing us to achieve a large number of tests. In order to focus our efforts to develop a thorough but practical testing regimen for the sample software, SVA, we modify the original SVA software to omit the portions concerned with input and output readings from the screen, thereby concentrating our effort on interrogating the logical operations of the program. The oracle software after modification reads the randomly sampled input data directly from input data files. For the format of input data file, refer to Appendix 5.4.

- 001 SVA program Input Reading

In SVA, the evaluations are conducted continuously, that is, if no terminating command is received, the next sets of input data should be evaluated using the values of all the parameters obtained from the last evaluation process. In order to ease the recording the flowpath of each test case, in the 001 SVA program, we purposely break the continuous process into single processes. But due to the continuity in the oracle SVA program and we have to compare results from both programs for correctness checking, we have to make sure both programs are under the same situation before any single evaluation process starts. To achieve this goal, we record all the intermediate parameter values after each single evaluation process done by the oracle SVA

program. The 001 SVA program will read the same input data as the oracle SVA program, plus all the intermediate parameter values recorded from execution of the oracle SVA program. The way this idea is achieved is that during execution of the oracle SVA program, a new input data file, including both the actual input data and the intermediate parameter values, is generated, ready to be converted to 001 SVA readable input data file. Refer to Appendix 5.5 for the format of such files (“oo#.dat”).

In 001, the data structures are formulated by TMaps. Thus, in order for the 001 SVA program to understand the input data, the files containing input data and intermediate parameter values have to be converted to OMaps, where the TMaps are stored. The conversion is completed by a program, named “make_complete_ins_from_file” constructed under 001 CASE Tool. The TMaps created for this purpose are described in Appendix 5.6. The resulting input files for 001 SVA program are named as “#.complete_in.omap”.

In the 001 Tool introduction section, we have mentioned Datafacer, a run-time system that automatically generates a user interface based on the data description in the TMap. It is a very helpful tool during normal testing and simulation exercises. But in this study, when we are dealing with a huge number of test cases, this graphic interface provided automatically by 001 Tool Suite takes much longer time than we could afford. After completing about one thousand test cases, it becomes the very bottleneck to our test task. Again, we have to remove this graphic interface. Some work is done within 001 Tool in order to avoid the interface generating step, which leads to 001 SVA program’s direct reading of its input data files.

4.5.2.1. Output Recording

In order to make test results checking automatic, instead of using on screen displaying of the results, the results from both oracle and 001 program are documented into output files. By uniform the output files format, it is very easy to compare testing results from 001 program with the corresponding oracle.

- Oracle Output Recording

The original oracle SVA program has an on screen result display as described in Appendix 5.3. With the aim of automatic testing result checking, the outputs are directed to output files named “oracle#.dat”. Each “oracle#.dat” file corresponds to one “i#.dat” file (the input data file). Every output screen has a section in “oracle#.dat” file. Each section includes warning and information messages, name, value and status of all sensors (normal sensors and PAMI sensors). Later on, these files will be used as standard to check if those output files from 001 SVA program are correct. Refer to Appendix 5.7 for format of “oracle#.dat” files.

- 001 SVA program Output Recording

001 SVA program reads the input data sets directly from “#.complete_in.omap” files. The resulting output are directed into files named “output#.#”, which have exactly the same format as “oracle#.dat” presented in Appendix 5.7.

There is yet another problem in comparing “oracle#.dat” and “ouput#.#” though they are of the same format. For the oracle SVA program, the algorithm works continuously. It processes all the input data sets in one input file “i#.dat”, with all the corresponding results to all the input data sets in one input file recorded in one output file “oracle#.dat”. But for 001 SVA program, it evaluates one input data set each time. This means we have to merge all the output files from 001 SVA program corresponding to one oracle SVA program into one file so that we can easily compare two output files of the same format and corresponding to the same group of input data sets. This work is done a small program written in PERL, “output2o.pl”. The merge output files

from 001 SVA program are named as “o#.dat”. Now “o#.dat” from 001 SVA program can be compared directly to “o#.dat” from oracle SVA program.

4.5.2.2. Automatic Output Comparison

Now we have obtained output files from both programs with the same format. A simple program written in C under DOS named “com_sva.c” helps us to complete the automatic testing process. This program simply compares oracle and experiment output files line by line. The output of this program is named “result#.dat”. If there is no difference, which means the tests are successful, we will see “There are totally 0 differences.” in “result#.dat”. Otherwise, the differences are listed in “result#.dat”.

4.5.3. Regression Testing

In most of the Software Reliability Growing Models (SRGMs), it is assumed all the detected errors are removed immediately without introducing new defects. In this software testing and reliability estimation study, due to the nature of the software under consideration, the perfect error removal becomes a requirement instead of an assumption. It is irresponsible to say that we assume we have got rid of the encountering error without introducing new error. We have to approve this is true. The technique we apply here is a very simple and traditional one, i.e., after each modification, we test the modified software again by feeding all the previous used input data to it. Unless the modified software behaves exactly the same as the oracle software in terms of all the input data, we will keep modifying the software toward satisfaction. As all the testing process is automatic, the regression testing process is very easy to perform as all the input data files (for both oracle SVA program and 001 SVA program), oracle output files are already available. We only need to rerun the 001 SVA program with all the existing input files and compare the output files with oracle files again.

4.5.4. Testing Results

We generated 198,321 test cases in total, out of which, 133,093 are unique. Fifty errors are detected through out the whole testing process. All of the errors are corrected immediately and regression testing is performed. In another word, for all the 133,093 unique input data sets, the 001 SVA program behaves exactly the same as the oracle SVA program, which, under our standard, means the 001 SVA program is perfectly correct in terms of the 133,093 unique input data sets. For an overview of all the encountered errors, refer to Appendix 5.8.

4.6. Reliability Estimation

4.6.1. Required Information

It is very important to make good approximation about the underlying software reliability after completing some tests on it. Otherwise, we will not be able to know when the software is safe enough so that we can stop testing. Only by comparing the reliability estimation result with the safety requirement of the target software, are we able to decide the further action to take. Before we make the reliability estimation for 001 SVA program, let us make a summary about the information needed.

For nodal coverage based reliability estimation approach, we need:

- A list of unique nodes on SVA program⁴;
- Visiting frequency of each node during testing;

For flowpath coverage based reliability estimation approach, we need:

- A list of unique flowpaths that have been tested;
- Visiting frequency of each flowpath during testing;
- Error information from the tested flowpaths;

In the following sections, details about how we get the above information from 001 SVA testing process and use it to estimate reliability of 001 SVA program. Though, in this thesis, the only the final reliability estimation results are presented, the same reliability techniques can be applied to the target software at any point in the middle of testing process. In reality, the testing and reliability estimation tasks should be applied to the underlying software interchangeably and continued until the estimated reliability level meets the requirement.

4.6.2. Flowpath Recording in 001

As has been defined before, a flowpath on a program is an ordered sequence of nodes on the control flowgraph of the software. Thus, only those decision making statements in a program are necessary to identify which flowpath of the control flowgraph has been taken upon an execution. Those processing statements, which make no decision, are surplus in terms of differentiating flowpaths. Within 001, in order to save storage space and processing time, most of the processing or functional statements or nodes are ignored while recording flowpaths during software execution. Minimum information is kept to remember the tested flowpaths.

4.6.2.1. The Nodes in 001

In 001, all the functions are defined as FMaps. Due to this reason, we use node instead of statement in this section. All the nodes within 001 can be divided into two types: decision node and functional node. To use 001 language, all the decision nodes have at least two children nodes. The fundamental control structure between a parent decision node and its children nodes are OR (O). After executing a decision node, one and only one of its children nodes (called alternative nodes of their parents) will be executed. It is obvious that only by remembering all the decision nodes and their chosen alternative nodes along a path, this path is uniquely recorded. For all the other nodes, since there is no decision to make, or, there is only one way to go from one such node, we always know that they are executed as long as their most recent previous alternative nodes have been executed. We call these nodes as functional nodes, meaning there is no decision making involved.

Though, there is no need to record any of the functional nodes, we feel it essential in recording a special group of them. Except decision nodes and their chosen alternatives, we also record the first node of every FMaps along the execution path. The reason is that if we only record the name of the decision making nodes and their alternative nodes that are triggered by a test case, we will not be able to know which FMaps they belong to. This is the same as whenever a decision node is recorded, besides its name, its residing subroutine name is recorded along. Thus, the information we get for one flowpath is as:

```
FMap head1
[decision node 1 → alternative1
```

⁴ It is required, in the nodal coverage reliability estimation approach, that all the nodes are tested at least once.

```

decision node 2 → alternative2
.....
]
FMap head2
[decision node 1 → alternative1
decision node 2 → alternative2
.....
]
.....

```

The recorded flowpaths are named as “#.decpaths.omap”.

4.6.2.2. Feasible Paths — Cutting Extra Iterations of Loop Structures

To make the testing reliability estimation approaches feasible, we limit the number of iterations of any repetitive structures. For most of the looping structures, they are possible to be expanded more than twice. In this study, we only want to examine a looping structure with zero, one and two iterations during execution. In another word, if a loop is repeated more than twice during a test case, we will treat it the same as though it has been through the loop twice. A program named “cut_iterations” within 001 is built to cut extra looping structure iterations from flowpaths recorded out of testing process. The input of this program is a text file containing the directory and names of the flowpath files (“#.decpaths.omap”) that need to be processed. Every flowpath file will be scanned by “cut_iterations” and the extra iterations cut. The simplified flowpaths files with at most two iterations of each looping structures are named “#.decpaths.omap.cut”. A new text file including all the simplified flowpath file names is output as a result.

Figure 4-3 Cut Extra Iterations of a Looping Structure

4.6.3. Reliability Estimation Process Overview

In this section, we present the 001 SVA program reliability estimation process as well as results according to both complete nodal coverage and feasible flowpath coverage approaches.

4.6.4. Complete Nodal Coverage Approach

In this approach, it is required that all the nodes⁵ are tested at least once so that each node has some testing evidence upon which the unreliability of each node can be estimated. In this project, the sample program 001 SVA is tested randomly in the beginning. Then a tool is made under 001 to estimate the visiting frequency of every node within 001 SVA program. When it is found that there are only a small amount of nodes that have not been visited, for each of the nodes that have zero visiting frequency during the passing tests, special investigation is performed. Eventually all the existing nodes that do not belong to 001 built in library are tested to some extent and the unreliability is estimated accordingly.

4.6.4.1. Node Visiting Frequency and Uncovered Nodes

To check out the nodes that have not been covered and the visiting frequency of the already tested nodes, a tool is developed under 001. The name of the tool is “search_uncovered_simple”⁶. As inputs, it takes in both the static un-expanded 001 FMaps of the target software and the recorded flowpaths of every test case. The methodology of the tool program is described in chapter **Error! Reference source not found.**. The output of the program is a list of the nodes of the target software with number of visits and visiting frequency attached after the name of the nodes.

We use the tool program “search_uncovered_simple” after 78,717 test cases⁷. Before that, we keep track of the speed of increase of the new detected flowpaths. While 78,717 is already a big number but the unique flowpaths are still showing up with a steady speed, we decide to check out if all the nodes have been covered and if the new detected flowpaths are only new combinations of the already tested nodes. As result of the node visiting frequency calculation, we find 16 unvisited nodes after the 78,717 test cases. Since 16 is not a big number, we manage to investigate them each by each. They are divided into two groups after the investigation. The nodes in the first group are the nodes that can only be reached by very special testing input data sets. They are designed to deal with very special cases, e.g. a wrong operator input value. It is almost impossible to reach those nodes accidentally by randomly sample data. The logic behind each node in this group is studied carefully and purposely designed input data are applied to 001 SVA to test these nodes. We find that 14 out of the 16 unvisited nodes belong to this group. There are 2 uncovered nodes left. They are the extra nodes which we decide can never be reached by any input data and thus do nothing good to the program. They end up to be deleted from the 001 SVA program. For details about all the 16 unvisited nodes after 78,717 tests on 001 SVA program, turn to Appendix5.10. We find 3 errors from the 14 nodes in the first uncovered node groups tested by the 84 test cases. Compare with 9 errors that are found from the previous 78,717 tests, the error detection efficiency is very high. Thus this is a very capable method to spot errors. For details of the errors, see the next section, where the software reliability is estimated based on flowpath coverage. To estimate software reliability, the error information is

⁵ If the software is developed under 001 Tool Suite environment, it is not required that all the 001 built in library nodes are tested to achieve complete nodal coverage. The reason is that all the 001 built in nodes have been through thorough tests when 001 Tool was built, plus all the built in FMaps within 001 are very simple.

⁶ For details about “search_uncovered_simple”, refer to Appendix5.9.

⁷ Test case number 0.1 to test cases number 1999.40.

not needed because any detected error should be fixed immediately under safety-critical requirement.

In 001 SVA program, there are totally 433 SVA nodes. Refer to Figure 4-4 for number of visits to SVA nodes. Notice that 001 internal nodes are not included.

Figure 4-4 Number of Visits to SVA Nodes after 198,321 Tests

4.6.4.2. Testing Results, from Nodes Point of View

Overall, we test 001 SVA program 198,321 times, out of which 133,093 tests are unique. There are 1063 nodes, among which 432 nodes are created by the programmer. The 1063 nodes have been tested 38,013,756 times through unique test cases (the test cases with unique input data sets); 27,007,494 tests hit on the 432 SVA nodes and $(38,013,756 - 27,007,494) = 11,006,262$ hit on SVA built-in nodes. The average number of visits to SVA nodes is $(27,007,494/432) = 62,517.3$.

12 errors are found during the whole testing process. 3 of them, as described in the last section, are detected by the input data designed to cover the untested nodes (testcase2000.1 ~ testcase2010.20, 84 test cases, inputs generated manually); 9 of them are identified before the process (testcase0.0 ~ testcase1999.40, 78,717 test cases, inputs generated manually and automatically); 0 error is found after the 84 specially designed test cases.

4.6.4.3. Reliability Estimation from Nodal Coverage Approach

We have already discussed the formulas to estimate software reliability based on nodal testing results. By using the results presented in the previous section, we can estimate SVA reliability at the end of the 198,321 tests accordingly. The same reliability estimation method can be performed at any moment during the testing process. If there has been any modification since

the last reliability estimation, the reliability estimation is a re-calculation since the underlying reliability level has been increased due to the perfect removal of errors. If there has been no new error found since the last estimation, the reliability estimation process is an update. The underlying reliability has not been changed. It is our knowledge about the target software that has been changed due to the more tests performed on it.

According to the derivation given in chapter **Error! Reference source not found.**, by using uniform prior unreliability distribution for each node, the unreliability probability distribution of node i_n that has been tested s times without any error is given by:

$$f(\theta'_{i_n}) = \frac{(1 - \theta'_{i_n})^s}{B(1, 1 + s)}$$

The average of the unreliability θ'_{i_n} can be calculated as:

$$\overline{\theta'_{i_n}} = \int_0^1 f(\theta'_{i_n}) d\theta'_{i_n} = \frac{1}{2 + s} \quad \text{Eq. 4-11}$$

s is the number of tests applied on node i_n . For each of the unique SVA node, we calculate its unreliability as Eq. 4-11. The unreliability values are shown in the following figure.

⁸ This probability is obtained through Bayesian updating method with uniform prior probability distribution for θ'_{i_n} , i.e. $\int_0^1 f(\theta'_i) = 1$. It is definitely true that a much better informed prior distribution can be used. But in this study, no solid prior distribution has been found for θ'_{i_n} .

Figure 4-5 SVA Unique Node Unreliability

As seen in Figure 4-5, the arithmetic average unreliability of all the unique SVA nodes is 1.575E-05. The average number of unique SVA nodes visited per execution in SVA is the total number of visits to all the unique SVA nodes, i.e., 27,007,494, divided by the number of total unique tests performed, i.e., 133,093. Thus,

$$\bar{x}_{SVA} N_{SVA} = \frac{27,007,494}{133,093} = 202.92^9 \quad \text{Eq. 4-12}$$

The overall unreliability of the SVA nodes is:

$$\theta_{n(SVA)} = 211.58 \times 1.575 \times 10^{-5} = 3.196 \times 10^{-3} \quad \text{Eq. 4-13}$$

The overall unreliability of SVA software is the average unreliability of the SVA nodes and the average unreliability of the built-in 001 nodes. We assume all the 001 nodes are perfectly reliable, thus,

$$\theta_n = \frac{27,007,494}{38,013,756} \times \theta_{n(SVA)} + \left(1 - \frac{27,007,494}{38,013,756}\right) \times \theta_{n(001)} = 2.27 \times 10^{-3} \quad \text{Eq. 4-14}$$

The 001 SVA reliability, according to nodal coverage reliability estimation method with uniform prior unreliability distribution, is:

$$R_n = 1 - \theta_n = 1 - 2.27 \times 10^{-3} = 0.997733 \quad \text{Eq. 4-15}$$

So far we have presented the process and result of unreliability estimation for SVA program. It is very straightforward to understand and simple to perform. For this specific software, SVA, we still have not found a good prior distribution to be incorporated into the unreliability Bayesian updating process. The only prior information we obtained for SVA, as will be discussed in the flowpath coverage approach session, is an average flowpath unreliability value, 0.01. For any software system, if there is a good prior distribution for the nodal unreliability, it is always better to use it. In this case, with only the average unreliability value, we suggest a shortcut to approximate the value. If we look at the software as one flowpath, its unreliability can be updated as:

$$\theta_{uni} = \frac{p_{II} (1 - p_{f,II})^s P e_0}{p_{II} (1 - p_{f,II})^s P e_0 + (1 - P e_0)} p_{f,II}$$

Eq. 4-16

⁹ The average number of node visited per test, 211.58 is greater than the total number of unique SVA nodes, 202. The reason is that one unique SVA node may be visited several times per execution.

If we take uniform prior, which means $Pe_0 = 0.5$ and substitute $p_{II}(1 - p_{f,II})^s$ with x , $p_{f,II}$ with y , we get:

$$\theta_{uni} = \frac{xy}{x+1}$$

Eq. 4-17

For SVA the average number of visit to a flowpath, the average s is greater than 100, which gives $x \ll 1$, we get:

$$\theta_{uni} \approx xy$$

Eq. 4-18

Now let us look at the situation where the average prior unreliability $Pe_0 \ll 1$. We have:

$$\theta_{non-uni} = \frac{p_{II}(1 - p_{f,II})^s Pe_0}{p_{II}(1 - p_{f,II})^s Pe_0 + (1 - Pe_0)} p_{f,II} = \frac{xy}{x + \frac{(1 - Pe_0)}{Pe_0}} \approx xyPe_0 = \theta_{uni}Pe_0$$

Eq. 4-19

From Eq. 4-19, if we take the nodal coverage based unreliability estimation result with uniform prior distribution as θ_{uni} , the unreliability value with prior average value 0.01 can be simply estimated as $\theta_{0.01} = \theta_{uni}Pe_0 = 2.27 \times 10^{-5}$.

Despite the weakness of nodal coverage based reliability estimation method, it is still a useful technique when a large program is under consideration, in which case, the following demonstrated flowpath coverage based approach is far too difficult to perform. By observing the visiting frequency to every node during testing, we can decide which node is more important in terms of its probability of being visited during an execution. Thus, more attention can be given to those more important nodes. It is more efficient in putting more energy into the more significant nodes. Furthermore, as can be seen from the error data, by searching for the input data to cover those nodes that have not been tested after a long time testing, it is easier to capture some unexpected errors in the software. Some surplus nodes that may never do anything good to the software can be spotted during the process too.

4.6.5. Refined Complete Feasible Path Testing Approach

Now, let us turn to our second reliability estimation method, the refined complete feasible flowpath based reliability estimation method. This is a finer approach, which gives better estimation, but also takes more information from the software under consideration. For safety-

critical software, which is relatively simple¹⁰, this method is recommended compared to the nodal coverage based approach.

4.6.5.1. Truncate Extra Iterations of Looping Structures

With the aim of making the complete flowpath testing task practical, we need to limit the number of loop iterations. If we were to expand the control flowgraph first and identify the input data to each flowpath, we would expand the looping structures to at most two repetitions and choose input data according to each expanded flowpath. Since we adopt the opposite way to achieve the same flowpath coverage objective, i.e., we randomly sample the input data first and identify their triggered flowpaths during execution, we have to accept whatever flowpaths recorded during the testing process. Instead of doing the cutting extra loop repetitions in the first place, now we have to cut the extra loop iterations after they get recorded.

Figure 4-6 shows how different expansions of the same looping structures are treated. The zero, one and two-repetition of a looping structure are treated as unique expansions of the structure. But if any flowpath that contains more than two iterations of a loop, it is treated as though it has only two iterations of the loop.

Figure 4-6 Cut Iteration in Flowpaths

With the intention to cut extra iterations of every looping structure for every recorded flowpath, a tool program named “cut_iterations” under 001 is built. It takes in the file that contains the directory and names of the recorded flowpath files (“#.#.decpaths.omap”) that need to be checked for extra iterations. Each of the flowpath is scanned by the program and all the extra repetitions within the flowpath are truncated. A new smaller flowpath file (“#.#.decpaths.omap.cut”) is produced by the tool with all the extra iterations cut. After this step,

¹⁰ For “relatively simple software”, a significant percentage of all its feasible flowpaths should be able to be tested.

unique flowpaths will be identified from these new cut flowpath records. For details about “cut_iterations” tool, refer Appendix5.11.

4.6.5.2. Distinct Feasible Flowpath Identification

Now, the surplus repetitions have been cut, it is the time for us to identify the unique flowpaths that have been tested. Originally, a 001 tool is built to do this job. Every flowpath is compared by the tool with all existing recorded flowpaths. The basic idea is to go through the structure of every two flowpaths, comparing every corresponding node from both flowpath. If there is any difference, the two flowpaths are believed to be different. If a flowpath is different from all of its previous flowpaths, it is recognized as a unique flowpath. While every flowpath is about 40Kb and due to the overhead of 001 for its safety feature, the process turns out to be very time and memory consuming. Let alone the time taken, during the process of comparing 1597 flowpath files, the memory is overflow. In order to solve this problem, two changes are made.

First, in stead of comparing all the flowpath files once through, we divide the files into several groups. From each group, a distinctive set of flowpath files are identified. Apparently, the size of each group shrinks. Then, some of the groups are combined, after which, we have less number of groups. Again, a distinctive set of flowpath files are identified from each group. This process is continued until only one distinctive group of flowpath files is left.

The second method we use to solve the time and space problem is to switch from a 001 tool to Linux system tool. It is obvious that all the flowpaths are remembered according to the same format. As a result, it is not necessary to compare every two flowpaths node by node. As long as the two files are different, their structures must be different. As opposed to 001 program, Linux shell commands and several small programs written in PERL are used for the file comparison task. Refer to Appendix5.12 for the details.

Figure 4-7 Number of Test Cases vs. Number of Flowpaths, SVA

Figure 4-7 shows the flowpath identification result. We test 198,321 input data sets on 001 SVA program and 9,870 unique flowpaths are identified.

4.6.5.3. Count Visiting Frequency of Flowpaths

To estimate unreliability of every tested flowpaths, besides the group of unique flowpaths, we need to know how many times each flowpaths have been tested. Again, in order to take better usage of the system resource, we turn to PERL instead of 001. Also, since the number of flowpaths we are dealing with is very big, 198,321, the visiting time counting task is attacked step by step. The details of the count process are discussed in Appendix 5.13. The following is the picture showing the visiting frequency of all the tested 001 SVA flowpaths.

Figure 4-8 Visiting Frequency of Unique 001 SVA Flowpaths

4.6.5.4. 001 SVA Reliability Estimation Based on Feasible Flowpath Coverage

We have collected all the testing data for 001 SVA program. The reliability estimation method based on software feasible flowpath coverage is demonstrated on SVA. The Unreliability of the tested flowpaths and that of the untested flowpaths are different, hence are estimated separately. Then the two unreliability values are average based on their anticipated probability of being visited during on execution.

4.6.5.4.1. Estimate Unreliability of the Tested Flowpaths

For the tested flowpaths, we first estimate their unreliability one by one. Then the mean value of all the unreliability values is calculated. Different weights are given to different flowpaths based on their visiting frequency during the testing process.

4.6.5.4.1.1. Percentage of Type I Error in 001 SVA

Before estimating reliability, the probability values associated with the two error types are estimated. In this project, these values are approximated from the target software itself. It is also possible that they are evaluated from similar program, which in turn, provide more error data points.

12 errors are detected and corrected in 001 SVA program as a result of 198,321 test cases. There are 9 type I errors and 3 type II errors. For details of the errors found within 001 SVA program, see Appendix 5.8.

Except the 4th, 8th, 12th errors, all the other 9 errors are of type I. The uniform distribution is used as the prior distribution for p_I , the probability that an error belongs to type I, i.e., $f_0(p_I) = 1$. With the 1st error, which is of type I, the distribution is updated as:

$$f_1(p_I) = p_I f_0(p_I) = C_1 p_I, \quad C_1 = \frac{1}{\int_0^1 p_I dp_I} \quad \text{Eq. 4-20}$$

After the 3rd error, the distribution becomes:

$$f_3(p_I) = C_3 p_I^3, \quad C_3 = \frac{1}{\int_0^1 p_I^3 dp_I} \quad \text{Eq. 4-21}$$

After the 4th error, the distribution is updated as:

$$f_4(p_I) = C'_4 (1 - p_I) f_3(p_I) = C_4 (1 - p_I) p_I^3, \quad C_4 = \frac{1}{\int_0^1 (1 - p_I) p_I^3 dp_I} \quad \text{Eq. 4-22}$$

The same process continues until the last error is used to update the distribution. At this point, we get:

$$f_{12}(p_I) = C_{12} (1 - p_I)^3 p_I^9, \quad C_{12} = \frac{1}{\int_0^1 (1 - p_I)^3 p_I^9 dp_I} \quad \text{Eq. 4-23}$$

The mean value calculated from Eq. 4-23 will be used later to estimate flowpath unreliability. The mean value is:

$$\bar{p}_I = \int_0^1 f_{12}(p_I) p_I dp_I = \frac{\int_0^1 (1-p_I)^3 p_I^9 dp_I}{\int_0^1 (1-p_I)^3 p_I^9 dp_I} \approx 0.714 \quad \text{Eq. 4-24}$$

Because there are only two types of errors, the average probability that an error belongs to the second group $\bar{p}_{II}$ is:

$$\bar{p}_{II} = 1 - \bar{p}_I = \frac{6}{17} \approx 0.286 \quad \text{Eq. 4-25}$$

We have $\hat{p}_I = 0.714$ and $\hat{p}_{II} = 0.286$.

4.6.5.4.1.2. Probability of encountering Type II Error in 001 SVA

This probability is related only to type II errors. Thus, for $p_{f,II}$, the probability that an type II error is encountered during an execution given there is a type II error on the flowpath that is executed, there are only 5 data points.

The 4th error is the first encountered type II error and is detected during the 4th visit to the flowpath. Assuming uniform prior distribution for $p_{f,II}$, its distribution can be updated as:

$$f_1(p_{f,II}) = C_1' (1-p_{f,II})^3 p_{f,II} f_0(p_{f,II}) = C_1 (1-p_{f,II})^3 p_{f,II} \quad \text{Eq. 4-26}$$

$$C_1 = \frac{1}{\int_0^1 (1-p_{f,II})^3 p_{f,II} dp_{f,II}}$$

The 8th and 12th errors are found during the 2nd and 4th. visit to their corresponding flowpath, thus:

$$f_2(p_{f,II}) = C_2 (1-p_{f,II})^4 p_{f,II}^2$$

$$f_3(p_{f,II}) = C_3 (1-p_{f,II})^7 p_{f,II}^3$$

$$C_3 = \frac{1}{\int_0^1 (1-p_{f,II})^7 p_{f,II}^3 dp_{f,II}}$$

Eq. 4-27

Now we can get the average value for $p_{f,II}$ from Eq. 4-27 as:

$$\bar{p}_{f,II} = \int_0^1 f_3(p_{f,II}) p_{f,II} dp_{f,II} = \frac{\int_0^1 (1-p_{f,II})^7 p_{f,II}^3 dp_{f,II}}{\int_0^1 (1-p_{f,II})^7 p_{f,II}^3 dp_{f,II}} \approx 0.333 \quad \text{Eq. 4-28}$$

$$\hat{p}_{f,II} = 0.333 \quad \text{Eq. 4-29}$$

4.6.5.4.1.3. Estimate Tested Flowpath Unreliability by Mean Value in 001 SVA

In this section, we will use the simpler method to estimate unreliability for each tested SVA flowpath. In this method, only the average unreliability value is considered and updated as test proceeds. For a flowpath that has been tested t times error free, the formula to use is:

$$p_e^t = \frac{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0}{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0 + (1 - p_e^0)} \quad \text{Eq. 4-30}$$

We need to get a proper prior distribution for the probability that there is an error within a flowpath and take its average as p_e^0 . Efforts has been made to search for a proper prior for p_e^0 , which turn out to be not very satisfying. As a result, some rough historic data are used in this study obtained from the U.S. software industry. The purpose of presenting this prior distribution here is to give an idea of how to apply this flowpath coverage based reliability methodology into real use. There is enough space left for further exploring within this area.

In Caper Jones' "Applied Software Measure, Assuring Productivity and Quality"[**Error! Reference source not found.**] published in 1991, valuable software quality data are collected from some 4000 software projects developed between 1950 and 1990. The fact that we use data from his report tends to be too conservative due to such reasons as our target software is developed under 001 CASE tool and has been through thorough modular testing before this final integrated testing stage, etc. while all the reported data in the book are based upon average software systems.

In Jones' book, the size of a software program is always expressed in the form of function point rather than code lines or statements. Function points are a measure of the size of computer applications and the projects that build them. The size is measured from a functional, or user, point of view. It is independent of the computer language, development methodology, technology or capability of the project team used to develop the application. Besides all its other strong points, apparently it is a more effective tool to compare software applications from different roots. Since we do not want to dig into the details of function points, we use the following table, also obtained from [**Error! Reference source not found.**], to estimate how many function points are within our 001 SVA program.

Size, function points	Effort, staff- month	Schedule, month	Staff	Assignment scope, func- tion points	Production rate, func- tion points per person- month
40,960	81,920	96	853	48	0.50
20,480	30,118	73	413	50	0.68
10,240	8,192	55	149	69	1.25
5,120	2,994	42	71	72	1.71
2,560	1,099	32	34	78	2.33
1,280	405	24	27	75	3.16
640	149	18	8	80	4.30
320	55	14	4	80	5.85
160	20	11	2	80	7.95
80	8.1	6	1.35	64	9.82
40	4.35	4.50	1.0	40	9.19
20	2.56	2.56	1.0	20	7.81
10	1.50	1.50	1.0	10	6.64
5	0.89	0.89	1.0	10	5.64

Table 4-3 Table3.11 on [**Error! Reference source not found.**]

Effort, Schedule, Staff, Assignment Scopes, and Production Rates for Selected Sizes of Software Projects

In Table 4-3, the line with 40 function points describes 001 SVA program the best: only one person is involved in developing the software, spending about 4 months or so on it. Then from figure 3.21 in the same book, the U.S. average software defect potentials, if checking the point with 40 function points, we get a potential defect number of about 100. Given 001 SVA program has about 10,000 flowpaths¹¹, we conclude that about $10,000/100=100$ flowpaths have one defect. In another word, the probability that one flowpath has a potential error in it is about $1/100=0.01$. The potential defects here include all major sources that will be encountered in a software system: requirements bugs, design bugs, coding bugs, user documentation bugs, and bad fixes or bugs accidentally injected while repairing another defect. As can be seen, the potential defects considered into this 0.01 probability is more than there might exist within 001 SVA program, which also tends to give relatively conservative reliability estimation.

¹¹ This estimation is described later.

Figure 4-9 U.S. Average Software Defect Potentials

Now through the above described approximate reasoning, we get the average prior probability that there is an error on one flowpath is:

$$p_e^0 = 0.01 \quad \text{Eq. 4-31}$$

Plug Eq. 4-31 and $\hat{p}_I = 0.647$, $p_{f,II} = 0.333$ into Eq. 4-30, we get:

$$p_e^t = \frac{(1 - 0.647)(1 - 0.333)^t \times 0.01}{(1 - 0.647)(1 - 0.333)^t \times 0.01 + (1 - 0.01)} = \frac{0.353 \times 0.666^t}{0.353 \times 0.666^t + 99} \quad \text{Eq. 4-32}$$

Eq. 4-32 is applied to every tested flowpath to get the probability that there is an error (must be type II) on that flowpath. The unreliability of a flowpath can be calculated as:

$$\theta^t = p_e^t \times p_{f,II} = 0.333 \times \frac{0.353 \times 0.666^t}{0.353 \times 0.666^t + 99} \quad \text{Eq. 4-33}$$

The following graph pictures the unreliability of all the tested 9870 flowpaths.

Figure 4-10 Tested Flowpath Unreliability in 001 SVA Program, Average Value Approach

Visiting frequency plotted in Figure 4-8 is utilized to average the unreliability values pictured in Figure 4-10, we get the average unreliability value for the tested flowpath as

$$\bar{\theta}_{p,T} = 4.043 \times 10^{-5} \quad \text{Eq. 4-34}$$

4.6.5.4.1.4. Estimate Tested Flowpath Unreliability by Gamma Distribution

In this section, we shall use the more complicated methods described in chapter **Error! Reference source not found.** to estimate the average unreliability of the tested flowpaths. The formula to update the error existing probability distribution of a flowpaths tested t times is:

$$p_e = C_t \int_0^1 [p_e(1 - p_I)(1 - p_{f,II})]^t p_e f_0(p_e) dp_e \quad \text{Eq. 4-35}$$

where, C_t is the normalization constant. In this method, a prior distribution for p_e , rather than an average value is required. In most software reliability estimation or updating process, standard Beta distribution is the function form that is adopted most:

$$f_x(x) = \frac{1}{B(q,r)} x^{q-1} (1-x)^{r-1}, \quad 0 \leq x \leq 1.0 \quad \text{Eq. 4-36}$$

The average value of the Beta distribution with parameters q and r is:

$$\bar{X} = \frac{q}{q+r} \quad \text{Eq. 4-37}$$

With a mean value of 0.01, we choose $q = 1$ and $r = 99$ for prior distribution of p_e for every tested flowpath. Eq. 4-35 becomes:

$$p_e = C_t \int_0^1 [p_e(1-p_t)(1-p_{f,II})]^t p_e(1-p_e)^{98} dp_e \quad \text{Eq. 4-38}$$

Matlab is used to get the answer from Eq. 4-38 for every flowpath. The unreliability is again calculated as $\theta^t = p_e^t \times p_{f,II}$. The results are shown in Figure 4-11.

Figure 4-11 Tested Flowpath Unreliability in 001 SVA Program, Distribution Approach

The average tested flowpath unreliability from this approach is:

$$\bar{\theta}_{p,T} = 5.57 \times 10^{-6} \quad \text{Eq. 4-39}$$

4.6.5.4.2. Estimate the Number of Possible Flowpaths, $p_{visit,T}$ and $p_{visit,U}$ for 001 SVA

Since there is no theoretical solution to this problem, two experimental solutions are provided. In this section, results from the two approaches are presented. As we will see, the first approach does not work in our problem. It is the second method that gives us the final answer.

4.6.5.4.2.1. Unfeasibility of the First Experimental Solution

We have two methods to solve the total number of existing flowpath problem. The first one we apply to 001 SVA program is the simpler and more straightforward one. The number of unique flowpath vs. the number of test cases plot is drawn as follows.

Figure 4-12 Flowpath vs. Test Case of 001 SVA Program

Both the experimental plot and the fitting plot are shown in Figure 4-12. From the picture, the anticipated total number of test cases needed to test every flowpath at least once is smaller than the total number of tests we have performed on the target program. When the tests carried out is very close to the total number of tests needed to test all the flowpaths, the roughness of this approach may results in such paradox outcome.

4.6.5.4.2.2. Estimate from the Second Experimental Solution

Now, this method becomes our only possible solution to our problem. The number of visits vs. flowpath index plot is needed for the target software.

Number of Visits to Flowpath (0 ~ 4999), All Test Cases

Figure 4-13 Number of Visits vs. Flowpath Index for 001 SVA

According to Figure 4-13, the probability that a flowpath whose index number is among $(x, x + dx)$ will be visited upon next execution is:

$$P(x)dx = 0.0004e^{-0.0004x}dx \quad \text{Eq. 4-40}$$

The probability that a flowpath whose index is greater than 9,870, or an untested flowpath that will be visited upon next execution is:

$$P(X > 9,870) = e^{-0.0004 \times 9,870} \approx 0.01929 \quad \text{Eq. 4-41}$$

Thus, we get:

$$p_{\text{visit},U} = 0.01929, \quad p_{\text{visit},T} = 1 - 0.01929 = 0.98971 \quad \text{Eq. 4-42}$$

With the purpose of demonstrating the stableness of the results obtained from this method at different testing stages, we carry out the same estimation at different points along the testing process. Each time, similar plots to Figure 4-13 are drawn. The probability that a flowpath whose index number is beyond x will be visited upon next execution always has the form:

$$P(X > x) = e^{-ax} \quad \text{Eq. 4-43}$$

The results are listed in Table 4-4.

Total Test Case No	30717	38717	58717	78717	118321	158321	198321
Total Flowpath No	3716	4305	5378	6324	7655	8759	9870
a	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	0.0004
Predictions	P(X>9870)	0.7190%	0.7190%	0.7190%	0.7190%	0.7190%	1.9293%
	P(X>10000)	0.6738%	0.6738%	0.6738%	0.6738%	0.6738%	1.8316%
	P(X>12000)	0.2479%	0.2479%	0.2479%	0.2479%	0.2479%	0.8230%
	P(X>13000)	0.1503%	0.1503%	0.1503%	0.1503%	0.1503%	0.5517%
	P(X>14000)	0.0912%	0.0912%	0.0912%	0.0912%	0.0912%	0.3698%

Table 4-4 Predict Flowpath Visiting Probability along the Testing Process

The data in Table 4-4 shows the persistency of the estimation results. Before the last column, the estimations of a are the same along the testing process.

4.6.5.4.3. Estimate Unreliability of the Untested Flowpaths

From the estimation result for $p_{visit,T}$ above, we notice that we the probability that a flowpath that have not been tested will show up in a random execution of the software is less than two percent, or we have tested the majority of the software. This says we could use the intrinsic reliability is the same as the observed reliability. With this on hand, rather than going through the complicated process to estimate unreliability of the software, we can take a much shorter cut. The unreliability of the untested flowpaths can be approximated by the intrinsic unreliability of the tested flowpath without removal of the detected defects. More specifically, we set the unreliability of the flowpaths that have type I errors on them to one; with type II errors to $p_{f,II}$ and re-estimate unreliability value of the tested flowpaths as before. The new unreliability value, with unreliability of all the faulty flowpaths unmodified, for the tested flowpaths, can be used to approximate the unreliability of the tested flowpaths.

The unreliability value estimated by average value approach is 2.2023E-04 and by Gamma distribution is 3.41E-04.

4.6.5.4.4. Estimate Average Unreliability of the Whole Software

Finally, we can put the tested and untested parts together and estimate the overall unreliability of the target software. Unreliability values from the two parts are averaged with their respective probabilities of being visited during an execution as the weights. The total unreliability is:

$$\theta_p = p_{visit,T} \bar{\theta}_{p,T} + p_{visit,U} \bar{\theta}_{p,U} \quad \text{Eq. 4-44}$$

The data from both average approach and the Gamma distribution approach are listed in Table 4-5.

Method Name	$p_{visit,T}$	$\bar{\theta}_{p,T}$	$p_{visit,U}$	$\bar{\theta}_{p,U}$
Average	98.071%	4.04E-5	1.929%	2.20E-4
Gamma Distribution	98.071%	5.57E-6	1.929%	3.41E-4

Table 4-5 Unreliability Data from Flowpath Coverage Based Approaches

Use the above data, we the 001 SVA program unreliability estimation obtained from the average approach is:

$$\theta_p = 98.071\% \times 4.04 \times 10^{-5} + 1.929\% \times 2.20 \times 10^{-4} = 4.39 \times 10^{-5} \quad \text{Eq. 4-45}$$

The corresponding reliability value is:

$$R_p = 1 - \theta_p = 99.99561\% \quad \text{Eq. 4-46}$$

From Gamma distribution approach,

$$\theta_p = 98.071\% \times 5.57 \times 10^{-6} + 1.929\% \times 3.41 \times 10^{-4} = 1.20 \times 10^{-5} \quad \text{Eq. 4-47}$$

The corresponding reliability value is:

$$R_p = 1 - \theta_p = 99.9988\% \quad \text{Eq. 4-48}$$

4.6.6. Results Summary

Manually and automatically (by means of random sampling), we have performed 198,321 tests on 001 SVA program. Among these tests, 133,093 unique input data sets are used. 15 errors are detected and removed during the process, among them there are 10 type I errors and 5 type II errors.

432 SVA (non-001-built-in nodes) nodes are identified from the program. The unreliability estimated from this approach is 2.27E-5.

9870 unique flowpaths are recognized on 001 SVA program. The probability that an error belongs to type I group is $p_I = 0.714$ and belongs to type II group is $p_{II} = 0.286$. The probability that a type I error is encountered during one execution is $p_{f,I} = 1$ and that for a type II error is $p_{f,II} = 0.333$. The probability that an untested flowpath is encountered upon next execution is $p_{visit,U} = 1.9293\%$; and for a tested flowpath, it is $p_{visit,T} = 98.0707\%$. From the average flowpath coverage approach, with 0.01 average prior unreliability for each flowpath, the unreliability of the tested flowpaths is $\theta_{p,T} = 4.04 \times 10^{-5}$ and that of the untested flowpaths is $\theta_{p,U} = 2.20 \times 10^{-4}$. The overall unreliability value is $\theta_p = 4.39 \times 10^{-5}$ with the corresponding reliability value $R_p = 99.9956\%$. From the Gamma distribution flowpath coverage approach,

with $f_0(\theta) = C(1 - \theta)^{98}$ prior probability distribution for unreliability of every flowpath, the unreliability of the tested flowpaths is estimated as $\theta_{p,T} = 5.57 \times 10^{-6}$ and that of the untested flowpaths as $\theta_{p,T} = 3.41 \times 10^{-4}$. The overall unreliability is estimated as $\theta_p = 1.2 \times 10^{-5}$. The corresponding reliability is $R_p = 99.9988\%$.

5. Appendix

5.1. Arithmetic Average Vs. Geometric Average

Definition of arithmetic average:

$$\bar{a} = \frac{w_1^a a_1 + w_2^a a_2 + \dots + w_n^a a_n}{w_1^a + w_2^a + \dots + w_n^a} \quad \text{Eq. 5.1-1}$$

Definition of geometric average:

$$\bar{g} = \sqrt[w_1^g + w_2^g + \dots + w_n^g]{g_1^{w_1^g} \times g_2^{w_2^g} \times \dots \times g_n^{w_n^g}} \quad \text{Eq. 5.1-2}$$

How do we decided which average value should be used and what is the difference between the above two formulas? The arithmetic average is relevant anytime several quantities add together to produce a total. It answers the question, “if all the quantities had the same value, what would that value have to be in order to achieve the same total?” In the same way, the geometric average is relevant any time several quantities multiple together to produce a product. The geometric average answers the question, “if all the quantities had the same value, what would that value have to be in order to achieve the same product?” With the intention of deciding which method we should use in the nodal coverage based software reliability estimation process, we must know whether the product or the summation is relevant to our problem.

Assuming independent reliability value for each node on one path, the overall path reliability can be expressed in terms of reliability of each node on the path as:

$$\theta = R_1 \times R_2 \times \dots \times R_n \quad \text{Eq. 5.1-3}$$

Expressed in unreliability, Eq. 5.1-3 becomes:

$$\theta = 1 - R = 1 - (1 - \theta_1)(1 - \theta_2) \dots (1 - \theta_n) \quad \text{Eq. 5.1-4}$$

If $\theta_i \ll 1$, $i = 1, 2, \dots, n$,

$$\theta \approx \theta_1 + \theta_2 + \dots + \theta_n \quad \text{Eq. 5.1-5}$$

For each input data set, whether the program will fail or not only depends on the path it will trigger within the program. Thus, from the point of view of each input data set, the software unreliability is roughly¹² the summation of the unreliability of the nodes on that path. Thus what we care about is the unreliability summation. According to the difference between arithmetic average and geometric average, apparently, arithmetic average is the proper formula we should use to calculate the software unreliability based on its nodal coverage.

5.2. SVA Input Data Sets Description

System parameters (file name: “GS_VALID.H”). The values in this category rarely changes over testing process. Only several sets of purposely-chosen numbers are chosen by the tester at the beginning and read by SVA program before the algorithm starts executing. There are three such parameters:

- System name;
- Total number of normal sensors;
- Total number of PAMI sensors;

There are only three numbers corresponding to the above three parameters in “GS_VALID.H” file. A typical “GS_VALID.H” file looks like: “0 4 2”, saying that the system name is “0”, there are “4” normal sensors and “2” PAMI sensors. The system name only has two possible values with “0” standing for “DPS system” and “1” standing for “DIAS system”. System name only affects the output format.

SVA process input parameters (file name: “i###.dat”). The values are generated randomly for algorithm testing purpose.

- Values of normal sensor readings¹³ ($s1, s2, \dots, sn$);
- Values of PAMI sensor readings¹⁴ ($p1, p2, \dots, pm$);
- Lower boundary for narrow range (lb);
- Upper boundary for narrow range (ub);
- Expected variation for normal sensors (ev);
- Instrumental uncertainty for normal sensors (iu);
- Lower boundary for wide range (lb_p);
- Upper boundary for wide range (ub_p);
- Expected variation for PAMI sensors (ev_p);
- Instrumental uncertainty for PAMI sensors (iu_p);
- Operator selection (op);

¹² “Roughly” is because we assume the independent relationship among the nodes on each path, which is not true in reality.

¹³ How many values for normal sensor readings are needed for one input set depends on “Total number of normal sensors” specified in “GS_VALID.H”.

¹⁴ How many values for PAMI sensor readings are needed for one input set depends on “Total number of PAMI sensors” specified in “GS_VALID.H”.

All the above numbers are sampled randomly according to section 4.5.1.2. The programs performing the random sampling, “data_sva.c”, “d_sva2.c” and “d_sva3.c”, are written in C.

5.3. Original Oracle SVA Interface

Running Generic Signal Validation Program ...							
Entering the input (RETURN: next item, ESC: run, DEL: delete, Q: quit)							
N_NAME	VALUE	STATUS	P_NAME	VALUE	STATUS		
1.SENSOR 1:	0	GOOD	5.P_SENSOR 1:	0	GOOD	INST UNC:	0
2.SENSOR 2:	0	GOOD	6.P_SENSOR 2:	0	GOOD	EXPE VAR:	0
3.SENSOR 3:	0	GOOD				N_MAX_RANGE:	0
4.SENSOR 4:	0	GOOD				N_MIN_RANGE:	0
Signal_Average:						P_INST_UNC:	0
Last_Valid_Signal:						P_EXPE_VAR:	0
						P_N_MAX_RANGE:	0
						P_N_MIN_RANGE:	0

Figure 5.3-1 Oracle SVA Program Initial Screen

Running Generic Signal Validation Program ...							
205 NORMAL_RANGE							
VALID							
PAMI							
Entering the input (RETURN: next item, ESC: run, DEL: delete, Q: quit)							
N_NAME	VALUE	STATUS	P_NAME	VALUE	STATUS		
1.SENSOR 1:	209	GOOD	5.P_SENSOR 1:	200	GOOD	INST UNC:	4
2.SENSOR 2:	207	GOOD	6.P_SENSOR 2:	198	GOOD	EXPE VAR:	3
3.SENSOR 3:	205	GOOD				N_MAX_RANGE:	400
4.SENSOR 4:	202	GOOD				N_MIN_RANGE:	200
Signal_Average:		205				P_INST_UNC:	8
Last_Valid_Signal:		0				P_EXPE_VAR:	6
						P_N_MAX_RANGE:	1000
						P_N_MIN_RANGE:	0
End of Signal Validation							

Figure 5.3-2 Oracle SVA Program Output Screen

5.4. Oracle SVA Input Data Set Files

Each row in input data file (“input#.dat” or “i#.dat”) is one set of input data for SVA testing purpose. If in “GS_VALID.H”, it is specified that there are four normal sensors and two PAMI sensors, the numbers within one row of the input data file give values of the following parameters accordingly, starting from the first number on the left:

3 4 400 200 6 8 400 200 195 200 197 207 187 201 0 ← one set of input numbers, a row in input data file “input#.dat” or “i#.dat”

- **3**: Expected variation for normal sensors;
- **4**: Instrumental Uncertainty for normal sensors;
- **400**: Upper boundary of narrow range;

- 200: Lower boundary of narrow range;
- 6: Expected variation for PAMI sensors;
- 8: Instrumental Uncertainty for PAMI sensors;
- 400: Upper boundary of wide range;
- 200: Lower boundary of wide range;
- 195 200 197 207: Four normal sensor readings;
- 187 201: Two PAMI sensor readings;
- 4: Operator selection;

5.5. Input Data File Including Intermediate Parameter Values

Each row in input data file (“oo1_#.dat” or “oo#.dat”) is one set of input data for 001 SVA program. The following is a typical row in such a file.

```
3 4 400 200 6 8 400 200 202 1 196 1 203 1 206 1 202 1 205 1 201
201 1 0 1 0 0 0 0 0 1 1 1 201 0
```

- 3: Instrumental uncertainty for normal sensors;
- 4: Expected variation for normal sensors;
- 400: Upper boundary for narrow range;
- 200: Lower boundary for narrow range;
- 6: Instrumental uncertainty for PAMI sensors;
- 8: Expected variation for PAMI sensors;
- 400: Upper boundary for wide range;
- 200: Lower boundary for wide range;
- 202: First normal sensor reading;
- 1: First normal sensor status¹⁵;
- 196: Second normal sensor reading;
- 1: Second normal sensor status;
- 203: Third normal sensor reading;
- 1: Third normal sensor status;
- 206: Fourth normal sensor reading;
- 1: Fourth normal sensor status;
- 202: First PAMI sensor reading;
- 1: First PAMI sensor status;
- 205: Second PAMI sensor reading;
- 1: Second PAMI sensor status;
- 201: Process representative;
- 201: Calculated signal;
- 1: Validation;
- 0: Valid_operator_permissive;

¹⁵ Sensor status recording notations: 1 for GOOD, 2 for BAD, 3 for SUSPECT, 4 for DEVIANT.

- **1**: Valid_pami;
- **0**: pami_operator_permissive;
- **0**: operator_select;
- **0**: selected_sensor_by_op;
- **0**: fault_select;
- **0**: selected_sensor_by_fault;
- **1**: range_status (5 types);
- **1**: current_range (normal/wide);
- **1**: pre_range (normal/wide);
- **201**: last_valid_value;
- **0**: operator selection;

5.6. TMaps Constructed for SVA 001 Input Data

Figure 5.6-1 TMaps Storing Input Data for 001 SVA Program

5.7. Output Files from Oracle SVA Program (“oracle#.dat”)

Here is a typical output file from oracle SVA program.

```

~~~~~ SVA output ~~~~~
Out Of Range in Normal_Range ←Warning information
SENSOR_1 202 GOOD ← Name, value and status of normal, sensor_1
SENSOR_2 200 GOOD
SENSOR_3 200 GOOD
SENSOR_4 200 GOOD
P_SENSOR_1 209 GOOD
P_SENSOR_2 199 GOOD
~~~~~
VALID
PAMI

```

204 WIDE_RANGE ← Result of the 1st. sets of inputs
End of Signal Validation ← End of processing of the 1st. sets of inputs
SENSOR_1 202 GOOD
SENSOR_2 200 GOOD
SENSOR_3 200 GOOD
SENSOR_4 200 GOOD
P_SENSOR_1 209 GOOD
P_SENSOR_2 199 GOOD
~~~~~  
Out Of Range in Normal\_Range ← 2<sup>nd</sup>. Sets of inputs starts to be processed  
SENSOR\_1 200 GOOD  
SENSOR\_2 204 GOOD  
SENSOR\_3 203 GOOD  
SENSOR\_4 200 GOOD  
P\_SENSOR\_1 208 GOOD  
P\_SENSOR\_2 189 GOOD  
~~~~~  
VALID
PAMI
198 WIDE_RANGE
End of Signal Validation ← End of processing of the 2nd. Sets of inputs
SENSOR_1 200 GOOD
SENSOR_2 204 GOOD
SENSOR_3 203 GOOD
SENSOR_4 200 GOOD
P_SENSOR_1 208 GOOD
P_SENSOR_2 189 GOOD
~~~~~
..... ← Starts to process the 3rd. inputs sets

5.8. SVA Errors Overview

Error Overview (Testcases 0.1 ~ Testcases 4999.40)						
FP IDX NO	ERROR TC	Time of Visits	Visiting Frequency	Visits Until Failure	IDX	
15	1.4 (1)	3	2.25E-05	1	1.4	1.4
16	1.5	7	5.26E-05	1	1.5	1.5
19	1.10	1	7.51E-06	1	1.10	1.10
112	9.8	21	1.58E-04	4	9.4	9.4
174	20.11	1	7.51E-06	1	20.11	20.11
178	20.16	1	7.51E-06	1	20.16	20.16
260	31.14	1	7.51E-06	1	31.14	31.14
43	47.14 (1)	4	3.01E-05	2	3.9	3.9
776	95.18	5	3.76E-05	1	95.18	95.18
3457	702.8	1	7.51E-06	1	702.8	702.8
6330	2001.1	2	1.50E-05	1	2001.1	2001.1
5753	2007.2(2)	11	8.26E-05	4	1651.2	1651.2

Unique TCs	133093	Error FPs	12	Typel Error
Total Errors	12	1-Error FPs	9	Typell Error
Typel Errors	9	2-Error FPs	0	
Typell Errors	3			

5.9. Calculate Node Visiting Frequency

“search_uncovered_simple” under OO1 is built to calculate how often every node of a given function map is visited through out a set of test cases. The inputs of this program include the static function map and the flowpaths of the test cases. The static function map is marked by the flowpaths. Every node of the map has a register. With one flowpath on hand, we find the corresponding register for every node on it, increase that register by one. After we finish processing all the flowpaths, every node on the function map should have a number greater or equal to zero. If the register of a node has zero value, it means its corresponding node has not been visited. If it has a value greater than zero, it has been visited. The visiting frequency is calculated as visiting times over total number of test cases. The output of this program is a set of strings. Each string starts with the path from the top node of given function tree to the node being considered, ends with the number of visits to this node and the visiting frequency of this node.

search_uncovered_simple:

Inputs:

FN0: Directory and name of the output file. The name of the file has an extension “strset.omap”.

PATH: Directory under which the static function map is located.

LIBNM0: Name of the library under which the static function map is built.

MAPNM0: Name of function map. For example, we can put in “svaTest_s”

TCSIDX: Name of the file that contains information of where to find all the flowpaths files.

Outputs:

A “*.strset.omap” files contains visiting time and frequency of every node under given function map.

Example:

Inputs:

FN0: /usr2/oo1/mt1/new_coveragedb/uncovered/svareults/try/0_499.

PATH: /usr2/oo1/mt1/sva_project/sva_simple_for_single_input/

LIBNM0: sva_simple_for_single_input

MAPNM0: svaTest_s

TCSIDX: /usr2/oo1/mt1/new_coveragedb/uncovered/svareults/try/0_499.svatcsidx.fn

The format of “0_499.svatcsidx.fn” is like:

6

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/decpaths0/

0_38¹⁶

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/decpaths0/

40_99

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/decpaths1/

100_199

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/decpaths2/

---

¹⁶ “search_uncovered_simple” will try to find a file named “0_38.coveragedb.omap” under directory “.../decpaths0”

200_299

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/decpaths3/

300_399

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/decpaths4/

400_499

Outputs:

A file named “0_499.strset.omap” under directory “.../try”. This file looks like:

```
svaTest_s-->svaGUltest_s-->k(DEC:input_suc_or_not|ALTNO:2)17 TIMES: 12018  
FRE: 0.0066677819  
svaTest_s-->svaGUltest_s-->DOsvaGUI(DEC:input_suc_or_not|ALTNO:1) TIMES:  
17877 FRE: 0.993332  
svaTest_s-->svaGUltest_s-->RunMenu_s--  
>SVA_process_s(DEC:outof_range_or_not|ALTNO:2) TIMES: 17877 FRE: 0.993332  
svaTest_s-->svaGUltest_s-->RunMenu_s--  
>range_error(DEC:outof_range_or_not|ALTNO:1) TIMES: 0 FRE: 020  
.....
```

5.10. Untested Nodes on 001 SVA Program after 78,717 Tests

There are 16 untested nodes after 78,717 tests (testcase0.0 ~ testcase1999.40) on 001 SVA program.

5.10.1. Nodes Needing to Be Tested, But Not Tested

There are 14 untested nodes that should be tested in 001 SVA program. 84 specially designed test cases (testcase2000.1 ~ testcase2010.20) are applied to 001 SVA program to test these nodes according to their logic. This input data picking is achieved manually, which means only a small amount of nodes can be tested in this way. If we applied “search_uncovered_simple” tool to 001 SVA program at a much earlier stage and found out more leftover nodes that need to be covered, we had to perform more random tests and checked out the uncovered nodes at a later stage when the number of nodes that have not been covered was small enough to handle manually. The list of the 14 uncovered yet need to be tested nodes is listed below. Each of them is followed by the logic in SVA 001 program that leads to that node and the index(es) of the test case(s) that cover that node. Finally the testing result of that node (if any error is found on that node) is presented.

[1] svaTest_s-->svaGUltest_s-->RunMenu_s--
>range_error(DEC:outof_range_or_not|ALTNO:1)

If there are range error(s) from one or more of the input sensor values(greater than the

¹⁷ Path from the function map’s top node to this node.

¹⁸ Number of visits to this node.

¹⁹ Visiting frequency of this node.

²⁰ This node has not been covered yet.

upper limit or lower than the lower limit, which give a range type of 5 from “range_comparasion”), this node is called.

- [2] svaTest_s-->svaGUltest_s-->RunMenu_s-->range_comparison-->k(DEC:type4_or_not|ALTNO:2)
Under the same situation as in [1], this node is called.
Covering test cases: 2000.1~2000.11;
1 errors been found from 2000.1.
- [3] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->output-->output_op_and_range-->k(DEC:out_of_range_or_not1|ALTNO:1)
This node is used to output the “out of range” warning message when a sparsely touched type of output format is used(We have two types of output format for SVA, one of them is almost used all the time and this one is seldom used). There is not a test case using this output format under the “out of range” situation.
- [4] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->operator_select-->op_check-->output-->output_op_and_range-->k(DEC:out_of_range_or_not1|ALTNO:1)
This is the same node as that in [3], except it is called from another path than in [3].
- [5] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->output-->output_op_and_range-->k(DEC:opsel_or_not1|ALTNO:1)*
This is another warning message “operator select” output node. There is no test case using this format under the operator selected situation.
- [6] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->operator_select-->op_check-->output-->output_op_and_range-->k(DEC:opsel_or_not1|ALTNO:1)
This is the same node as that in [5], except that it is called from another path than in [5].
Covering testcases: 2008.1~2008.20, 2009.1~2009.20, 2010.1~2010.20
- [7] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->operator_select-->operator_input_from_file-->clone1(DEC:overflow_or_not|ALTNO:1)
When operator has selected a sensor which does not exist, this node is called. For example, if we have 2 normal sensors and 4 pami sensors(their code names should be from 1 to 6, which means a valid input should be from 0, meaning no selection, to 6) and the operator selects sensor number 7, this node is called.
Covering testcases: 2001.1~2001.2
1 error has been found from 2001.1~2001.2.
- [8] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->valid_sig_from_nsensors1-->valid_sig_from_nsensors-->devcheck_fail-->more_than_one(DEC:one_maxdev_or_more|ALTNO:2)
When there are 2 or more normal sensors having the same greatest deviation, this node is called.
- [9] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->valid_sig_from_nsensors1-->valid_sig_from_nsensors-->devcheck_fail-->unsatisfied_deviation1-->clone1(DEC:one_or_more|ALTNO:2)
The same logic as that described in [8] would cause this node to be called.

- [10] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s--
 >valid_sig_from_psensors_or_not-->calculate_sig_from_psensors--
 >pami_devcheck_fail-->unsatisfied_deviation1--
 >clone1(DEC:one_or_more|ALTNO:2)
 This is the same node as that in [9], except called from a different node.
 Covering testcases: 2002.1~2002.2
- [11] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s--
 >valid_sig_from_psensors_or_not-->calculate_sig_from_psensors--
 >pami_devcheck_fail-->more_than_one(DEC:one_maxdev_or_more|ALTNO:2)
 When there are 2 or more pami sensors having the same greatest deviation, this node is called.
 Covering testcases: 2003.1~2003.2
- [12] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s--
 >valid_sig_from_nsensors1-->valid_sig_from_nsensors-->devcheck_fail--
 >clone1(DEC:less_than_three_pass|ALTNO:2)
 If “pass==3” when trying to select representative value from normal sensors, this node is called.
 Covering testcases: 2004.1
- [13] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s--
 >valid_sig_from_psensors_or_not-->calculate_sig_from_psensors--
 >pami_devcheck_fail-->clone1(DEC:less_than_three_pass|ALTNO:2)
 If “pass==3” when trying to select representative value from pami sensors, this node is called.
 Covering testcases: 2005.1
- [14] svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->fault_select--
 >failed_validation-->clone1(DEC:fault_sel_or_not2|ALTNO:2)
 This node is called when no valid sensor can be found through “last_valid_check1”. It seems not necessary since “last_valid_check1” can always find a sensor whose value has a deviation from “last_valid_value” less than 2000.
 Covering testcases: 2007.1~2007.2
 1 error has been found.

5.10.2. Unnecessary Nodes

We find two uncovered nodes in 001 SVA that are of no use and cannot be covered at all. They are deleted from 001 SVA program.

- a) svaTest_s-->svaGUltest_s-->RunMenu_s-->label_pause1--
 >clone1(DEC:label_pause1|ALTNO:1)
 When “reverse==1”, this node is called. It seems “reverse” does not need to be “-1” in any circumstances.
- b) svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->fault_select--
 >failed_validation-->f_sel_from_pami-->last_valid_check2--
 >last_fault_selected_sensor_value--
 >fault_select_value(DEC:pami_or_not|ALTNO:1)

If fault selected value is from pami, this node is called. This is a surplus node, since if “fault_selected” sensor is from pami, “last_valid_check2” cannot be called. Delete the whole decision node “pami_or_not” within “fault_select_value”.

5.11. Cut Extra Loop Repetitions

In order to make the Complete Path Testing feasible, we keep at most two repetitions for one loop structure. In this section, the process to cut extra repetitions for loop structures is illustrated.

5.11.1. Working Principle of “cut_iterations”

Figure 5.11.1-1 A Static Unexpanded 001 FMap Containing Looping Structure

The FMap shown in Figure 5.11.1-1 contains a looping structure 3-6-7-8-3. There is only one decision node, node 3. Each time node 3 is reached, a decision is made. If node 7 is chosen as the next executed node after node 3, the loop is entered. If node 6 is chosen as the next execution node after node 3, the loop is exited. We call node 3 the loop entrance node and node 6 the loop exit node. As in any 001 FMap, the static FMap is executed according to the order top to bottom and right to left. If there is no decision node in the program, every node will be executed according to this order.

- Directory: name of the directory where “Fn” is located.
20
/usr2/OO1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/
0.1.decpaths.omap.cut
0.2.decpaths.omap.cut
...
0.20.decpaths.omap.cut

- Outputs

- Fn.out: name of file that includes names of all the output flowpath files “#.#.decpaths.omap.cut”. It looks like:
- All the “#.#.decpaths.omap.cut” files listed in “Fn.out”.

5.12. Steps to Identify Unique Flowpaths

- The Steps

- Put all the flowpath files (“#.#.decpaths.omap.cut”) under current directory. List all the concerning files sorted by their size under this directory and pipe this information into a file — F1.
- Use a program written in PERL, “size.pl” to process the resulting file from step 1, F1. “size.pl” gives output file F2. F2 only contains file names from F1. Each line of F2 only contains names of files that have the same size. Two file names within one line are separated by a blank space.
- Use a program written in PERL, “diff3.pl” to process the resulting file from step 2, F2. “diff3.pl” groups the files listed in F2 into different flowpath groups. There are two output files from “diff3.pl”. One is “F2.diff” and the other is “F2.diff.idx”. In “F2.diff.idx”, every line includes a group of flowpath files, separated by blank spaces. In “F2.diff”, every line has only one file, which is flowpath of the earliest test case²¹ among its flowpath group. In “F2.diff.idx”, every line contains all the flowpath file names belong to the same flowpath group.
- If there are too many flowpath files, we can put them into different directories and repeat the same steps from 1 to 3 under each directory. After a distinct group of flowpaths are extracted from each directory, they can be mixed and step 1 through 3 repeated again. Eventually, a distinct group of flowpaths are extracted for all the recorded flowpaths.

---

²¹ How early a flowpath was recorded can be decided from its name. For example, a flowpath file named “4.1.decpaths.omap” must be recorded earlier than “20.3.decpaths.omap” and “4.2.decpaths.omap” must be recorded earlier than “4.10.decpaths.omap”.

Figure 5.12-1 Identify Unique Flowpaths Step by Step

- An Example

We have flowpath files “0.0.decpaths.omap”, “0.1.decpaths.omap.cut”, ... , “99.39.decpaths.omap.cut”, “99.40.decpaths.omap.cut” under directory “./decpaths0/” and flowpath files “100.0.decpaths.omap.cut”, “100.1.decpaths.omap.cut”, ... , “199.39.decpaths.omap.cut”, “199.40.decpaths.omap.cut” under directory “./decpaths1/”. We want to find out a group of distinct flowpaths out of the files under these two directories.

Under “./decpaths0”

- ***ls -S -l *.cut > index0_99.in*** [index0_99.in]
- ***perl size.pl index0_99.in index0_99.out*** [index0_99.out]
- ***perl diff3.pl index0_99.out*** [index0_99.out.diff][index0_99.out.diff.idx]
- ***awk '{print \$1}' index0_99.out.diff | xargs -i cp {} ../0_199***

Under “./decpaths1”

- ***ls -S -l *.cut > index100_199.in*** [index100_199.in]
- ***perl size.pl index100_199.in index100_199.out*** [index100_199.out]
- ***perl diff3.pl index100_199.out*** [index100_199.out.diff] [index100_199.out.diff.idx]
- ***awk '{print \$1}' index100_199.out.diff | xargs -i cp {} ../0_199***

Under “../0_199”

- ***ls -S -l *.cut > index0_199.in*** [index0_199.in]
- ***perl size.pl index0_199.in index0_199.out*** [index0_199.out]

- `perl diff3.pl index0_199.out` `[index0_199.out.diff][index0_199.out.diff.idx]`

5.13. Count Visiting Frequency of Flowpaths

- The Steps

- Put all the flowpath files (“*.decpaths.omap.cut”) under current directory. List all the concerning files sorted by their size under this directory and pipe this information into a file — F1.
- Use a program written in PERL, “size.pl” to process the resulting file from step 1, F1. “size.pl” gives output file F2. F2 only contains file names from F1. Each line of F2 only contains names of files that have the same size. Two file names within one line are separated by a blank space.
- Use a program written in PERL, “diff3.pl” to process the resulting file from step 2, F2. “diff3.pl” groups the files listed in F2 into different flowpath groups. There are two output files from “diff3.pl”. One is “F2.diff” and the other is “F2.diff.idx”. In “F2.diff.idx”, every line includes a group of flowpath files, separated by blank spaces. In “F2.diff”, every line has only one file, which is flowpath of the earliest test case²² among its flowpath group. In “F2.diff.idx”, every line contains all the flowpath file names belonging to the same flowpath group.
- If there are too many flowpath files, we can put them into different directories and repeat the same steps from 1 to 3 under each directory. After a distinct group of flowpaths is extracted from each directory, they can be mixed and step 1 through 3 repeated again. In addition, the new “*.diff.idx” file has to be expanded since it only contains the names of flowpath files within the new directory. A complete index file is needed in order to count how many times a flowpath is visited. We have built a program named “expand1.pl” in PERL to do this job. Eventually, a distinct group of flowpaths is extracted out of all the recorded flowpaths.
- The lines of final index file “FINAL.diff.idx” needs to be sorted according to when the first test case in this line being visited. To achieve this, we have to sort the final “FINAL.diff” file which contains only the first test case in the final index file in one line. This is done by “sort.pl” in PERL. Then the sorted version of “FINAL.diff”, which is “FINAL.diff.sort”, is expanded using the final index file “FINAL.diff.idx” and becomes “FINAL.diff.sort.exp”.
- Finally the complete index file “FINAL.diff.sort.exp” is used to count the number of visits to every tested flowpaths. The counting job is done by “pathfre.pl” in PERL.
- If we want to know how does the total number of distinct flowpaths increases as the total number of test cases increased, “count_diff.pl” can be used.

- An Example

We have flowpath files “0.0.decpaths.omap”, “0.1.decpaths.omap.cut”, ... , “99.39.decpaths.omap.cut”, “99.40.decpaths.omap.cut” under directory “../decpaths0/” and flowpath files “100.0.decpaths.omap.cut”, “100.1.decpaths.omap.cut”, ... , “199.39.decpaths.omap.cut”, “199.40.decpaths.omap.cut” under directory “../decpaths1/”. We

²² How early a flowpath was recorded can be decided from its name. For example, a flowpath file named “4.1.decpaths.omap” must be recorded earlier than “20.3.decpaths.omap” and “4.2.decpaths.omap” must be recorded earlier than “4.10.decpaths.omap”.

want to find out how many distinct flowpaths exist under these two directories and the number of visits to each distinct flowpath.

- **`ls -S -l *.cut > index0_99.in`**

[index0_99.in]

```
-rw-rw-rw- 1 yi oo1tool 40623 May 31 14:44 55.30.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 40550 May 31 14:45 63.11.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 40550 May 31 14:47 71.11.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 39983 May 31 14:44 53.30.decpaths.omap.cut
.....
```

Files of The Same Size

- **`perl size.pl index0_99.in index0_99.out`**

[index0_99.out]

```
55.30.decpaths.omap.cut
63.11.decpaths.omap.cut 71.11.decpaths.omap.cut
53.30.decpaths.omap.cut
.....
```

Files of The Same Size

- **`perl diff3.pl index0_99.out`**

[index0_99.out.diff]

```
55.30.decpaths.omap.cut
63.11.decpaths.omap.cut
53.30.decpaths.omap.cut
.....
```

**The Representative File
of the Flowpath Group**

[index0_99.out.diff.idx]

```
55.30.decpaths.omap.cut
63.11.decpaths.omap.cut 71.11.decpaths.omap.cut
53.30.decpaths.omap.cut
.....
```

Files Representing

- **`awk '{print $1}' index0_99.out.diff | xargs -i cp {} ../0_199`**
Copy files listed in "index0_99.out.diff" to directory "../0_199".
- **`cp index0_99.out.diff.idx ../0_199`**
Copy file "index0_99.out.diff.idx" to directory "../0_199".

Under "../decpaths1"

- **`ls -S -l *.cut > index100_199.in`**

[index100_199.in]

```
-rw-rw-rw- 1 yi oo1tool 39750 May 29 15:31 119.1.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 39685 May 29 15:30 103.1.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 39685 May 29 15:31 115.1.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 39679 May 29 15:30 111.1.decpaths.omap.cut
.....
```

Files of The Same Size

- **`perl size.pl index100_199.in index100_199.out`**

[index100_199.out]

```
119.1.decpaths.omap.cut
103.1.decpaths.omap.cut 115.1.decpaths.omap.cut
111.1.decpaths.omap.cut
.....
```

Files of The Same Size

- **`perl diff3.pl index100_199.out`**

The Representative File of The Flowpath Group

```
[index100_199.out.diff]
119.1.decpaths.omap.cut
103.1.decpaths.omap.cut
111.1.decpaths.omap.cut
.....
[index100_199.out.diff.idx]
119.1.decpaths.omap.cut
103.1.decpaths.omap.cut
115.1.decpaths.omap.cut
111.1.decpaths.omap.cut
.....
```

Files Representing The Same Flowpath

- **awk '{print \$1}' index100_199.out.diff | xargs -i cp {} ../0_199**
Copy files listed in "index100_199.out.diff" to directory "../0_199".
- **cp index100_199.out.diff.idx ../0_199**
Copy file "index100_199.out.diff.idx" to directory "../0_199".

Under "../0_199"

- **ls -l *.cut > index0_199.in**

[index0_199.in]

```
-rw-rw-rw- 1 yi oo1tool 30621 Jul 11 20:37 114.9.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 30612 Jul 11 20:34 53.27.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 30610 Jul 11 20:37 173.6.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 30610 Jul 11 20:34 85.31.decpaths.omap.cut
-rw-rw-rw- 1 yi oo1tool 30582 Jul 11 20:37 175.3.decpaths.omap.cut
.....
```

Files of The Same Size

Files from Directory "../0_100_199"

Files from Directory "../0_99"

- **perl size.pl index0_199.in index0_199.out**

[index0_199.out]

```
114.9.decpaths.omap.cut
53.27.decpaths.omap.cut
85.31.decpaths.omap.cut
173.6.decpaths.omap.cut
175.3.decpaths.omap.cut
.....
```

Files of the Same Size

- **perl diff3.pl index0_199.out**

[index0_199.out.diff]

```
114.9.decpaths.omap.cut
53.27.decpaths.omap.cut
85.31.decpaths.omap.cut
175.3.decpaths.omap.cut
.....
```

[index0_199.out.diff.idx]

```
114.9.decpaths.omap.cut
53.27.decpaths.omap.cut
85.31.decpaths.omap.cut
173.6.decpaths.omap.cut
175.3.decpaths.omap.cut
.....
```

Files Representing the Same Flowpath

[index0_199.out.diff][index0_199.out.diff.idx]

- **perl expand1.pl index0_199.out.diff.idx index0_99.out.diff.idx index100_199.out.diff.idx**

[index0_199.out.diff.idx.exp]

```
114.9.decpaths.omap.cut
53.27.decpaths.omap.cut
85.13.decpaths.omap.cut 173.6.decpaths.omap.cut 173.14.decpaths.omap.cut
175.3.decpaths.omap.cut 175.17.decpaths.omap.cut 175.21.decpaths.omap.cut
175.25.decpaths.omap.cut
.....
```

Extended Test Case Names

- ***perl sort.pl index0_199.out.diff***

[index0_199.out.diff.sort]

53.27.decpaths.omap.cut
85.13.decpaths.omap.cut
114.9.decpaths.omap.cut
175.3.decpaths.omap.cut
.....

Ordered According to Time of Testing

- ***perl expand1.pl index0_99.out.diff.sort index0_199.out.diff.idx.exp***

[index0_199.out.diff.sort.exp]

53.27.decpaths.omap.cut
85.13.decpaths.omap.cut 173.6.decpaths.omap.cut 173.14.decpaths.omap.cut
175.3.decpaths.omap.cut 175.17.decpaths.omap.cut 175.21.decpaths.omap.cut
175.25.decpaths.omap.cut
.....

Ordered According to Time of Testing

- ***perl pathfre.pl index0_199.out.diff.sort.exp***

[index0_199.out.diff.sort.exp.pf]

NAME: index0_199.out.diff.sort.exp.pf
BASE: index0_199.out.diff.sort.exp
TOTAL TESTCASES: 6717
TOTAL PATHS: 1709

53.27.decpaths.omap.cut 1
85.13.decpaths.omap.cut 3
175.3.decpaths.omap.cut 4
.....

Time of Visits

- ***perl count_diff.pl total0_199.in.sort index0_199.out.diff.sort index0_199.count***

[diff0_199.count]

1
2 1
3 2
4 3
.....

Total Number of Distinct Flowpaths

Total Number of Test Cases

6. References

- Cheu80 R. C. Cheung. "A user-oriented software reliability model", IEEE Trans. On Software Engineering, 6(2): 118-125, 1980
- Litt92 B. Littlewood and L. Strigini, "The risks of software," Scientific American Now pp. 62~75, 1992

- Lyu96 M. R. Lyu (Ed.), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, 1996
- Menz00 T. Menzies and B. Cukic, “When to test less”, *IEEE Software*, 17(5): pp. 107-112, 2000
- Menz02 T. Menzies and B. Cukic, “How many tests are enough?” In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, Volume II, 2002
- Musa83 Musa, John D., and Okumoto, K., “Software Reliability Models: Concepts, Classification, Comparisons, and Practice” *Electronic Systems Effectiveness and Life Cycle Costing*, J.K. Skwirzynski (ed.) NATO ASI Series, F3, Springer-Verlag, Heidelberg, pp. 395-424
- Sui98 Y. Sui, “Reliability Improvement and Assessment of Safety Critical Software ”, Massachusetts Institute of Technology, May 1998
- Triv01 K. Popstajanova and K. Trivedi, “Architecture based approach to reliability assessment of software systems”, *Performance Evaluation*, Vol. 45, No.2, June 2001