

Reliability Quantification of Nuclear Safety-Related Software

by

Yi Zhang

A thesis submitted to the Department of Nuclear Engineering in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Nuclear Engineering
at the Massachusetts Institute of Technology

February, 2004

© Massachusetts Institute of Technology, 2004. All Rights Reserved.

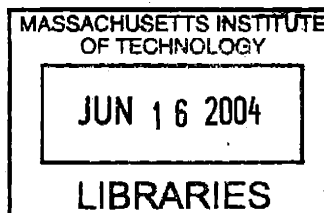
Signature of Author _____
Department of Nuclear Engineering

Approved by M. Golay _____
Michael W. Golay
Professor of Nuclear Engineering
Thesis Advisor

Accepted by G. Apostolakis _____
George Apostolakis
Professor of Nuclear Engineering

Accepted by _____
John A. Bernard, Jr.
Principal Research Engineer
Director, Nuclear Reactor Laboratory

Accepted by _____
Jeffrey A. Coderre
Chairman, Departmental Committee on Graduate Students



ARCHIVES

Reliability Quantification of Nuclear Safety-Related Software

by

Yi Zhang

Abstract

The objective of this study is to improve quality and reliability of safety-critical software in the nuclear industry. It is accomplished by focusing on the following two areas:

Formulation of a standard extensive integrated software testing strategy for safety-critical software, and

Development of systematic test-based statistical software reliability quantification methodologies.

The first step to improving the overall performance of software is to develop a comprehensive testing strategy, the gray box testing method. It has incorporated favorable aspects of white box and black box testing techniques. The safety-critical features of the software and feasibility of the methodology are the key drivers in determining the architecture for the testing strategy. Monte Carlo technique is applied to randomly sample inputs based on the probability density function derived from the specification of the given software. Software flowpaths accessed during testing are identified and recorded. Complete nodal coverage testing is achieved by automatic coverage checking. It is guaranteed that the most popular flowpaths of the software are tested.

The second part of the methodology is the quantification of software performance. Two Bayesian based white box reliability estimation methodologies, nodal coverage-based and flowpath coverage-based, are developed. The number of detected errors and the failure-free operations, the objective and subjective knowledge of the given software, and the testing and software structure information are systematically incorporated into both reliability estimation approaches. The concept of two error groups in terms of testability is initiated to better capture reliability features of the given software. The reliability of the tested flowpaths of the software and that of the untested flowpaths can be updated at any point during testing. Overall software reliability is calculated as a weighted average of the tested and untested parts of the software, with the probability of being visited upon next execution as the weight of each part.

All of the designed testing and reliability estimation strategies are successfully implemented and automated via various software tools and demonstrated on a typical safety-critical software application.

Thesis Supervisor: Michael W. Golay

Title: Reliability Quantification of Nuclear Safety-Related Software

Acknowledgements

Here is my opportunity to acknowledge some of those who made this work possible:

Professor Michael W. Golay, for not only giving me the opportunity to work on such an interesting project, but also for his advise and guidance over the last four and a half years; Professor George Apostolakis and Dr. John Bernard, for their valuable contributions in reading and discussions of the thesis; Professor Craig Carter, for his great help in completing this work.

Hamilton Technologies Inc. for its providing us with 001 system; Margaret Hamilton for her generous offer of the 001 Tool Suite; to Ronald Hackler for his technical guidance in using the wonderful features of 001 system; to Hannah Gold for her patient support for installation of 001 system.

My mother, father and sister, who have always been there for me and their never ending love.

My friends at MIT, for their great friendship, without what, this journey would have been very lonely.

Table of Contents

Abstract	3
Acknowledgements	4
Table of Contents	5
List of Figures	8
List of Tables	10
1. Introduction	11
1.1. Background	11
1.2. Objectives	12
1.3. Main Contributions	15
1.3.1. Grey Box Testing Technique	15
1.3.2. Nodal Coverage Based Bayesian Reliability Estimation Methodology	17
1.3.3. Flowpath Coverage Based Bayesian Reliability Estimation Methodology	18
1.4. Thesis Organization	19
2. The Integrated Formal Approach — DBTF (Develop Before The Fact)	20
2.1. Integrated Formal Approach	20
2.2. DBTF Process	22
3. Integrated Testing and Reliability Estimation Strategies	24
3.1. General Assumptions	24
3.2. Definitions	25
3.2.1. The Control Flowgraph	25
3.2.2. Flowpath	25
3.2.3. Feasible Flowpaths	26
3.3. Testing Methodology	27
3.3.1. Testing Strategy Overview	27
3.3.2. Software Testing Method Literature	27
3.3.3. A Mixed Testing Technique— Gray Box Method	34
3.3.4. Testing Details	40
3.4. Reliability Estimation Methodologies	52
3.4.1. Software Reliability Model Review	52
3.4.2. Two Reliability Estimation Methods Overview	67

3.4.3.	Complete Nodal Coverage Reliability Estimation.....	68
3.4.4.	Refined Feasible Flowpath Coverage Reliability Estimation.....	82
4.	Demonstration.....	134
4.1.	Introduction.....	134
4.1.1.	General.....	134
4.1.2.	Working Path Flow.....	135
4.2.	Sample Software — Signal Validation Algorithm (SVA).....	135
4.2.1.	SVA — the Algorithm.....	136
4.2.2.	Original Design Document of SVA.....	137
4.3.	001 CASE Tool.....	139
4.3.1.	DBTF.....	139
4.3.2.	001 System.....	140
4.3.3.	001 AXES Language.....	142
4.3.4.	Software Life Cycle in 001.....	146
4.4.	Capture SVA Specification into 001 TMaps and FMaps.....	147
4.4.1.	Formal Structures Building.....	147
4.4.2.	Specification Incompleteness and Inconsistency Captured.....	148
4.4.3.	Modular Testing.....	149
4.5.	Integrated Testing on SVA.....	150
4.5.1.	Test Data Generation.....	151
4.5.2.	Automatic Testing Process.....	159
4.5.3.	Regression Testing.....	163
4.5.4.	Testing Results.....	163
4.6.	Reliability Estimation.....	164
4.6.1.	Required Information.....	164
4.6.2.	Flowpath Recording in 001.....	165
4.6.3.	Reliability Estimation Process Overview.....	167
4.6.4.	Complete Nodal Coverage Approach.....	167
4.6.5.	Refined Complete Feasible Path Testing Approach.....	176
4.6.6.	Results Summary.....	193
5.	Discussions.....	195

6.	Conclusions and Future Work	202
6.1.	Conclusion	202
6.2.	Future Work	204
7.	Appendix	207
7.1.	Arithmetic Average Vs. Geometric Average	207
7.2.	SVA Input Data Sets Description	209
7.3.	Original Oracle SVA Interface	211
7.4.	Oracle SVA Input Data Set Files	212
7.5.	Input Data File Including Intermediate Parameter Values	213
7.6.	TMaps Constructed for SVA 001 Input Data	215
7.7.	Output Files from Oracle SVA Program (“oracle#.dat”).....	216
7.8.	SVA Error Details.....	217
7.9.	Calculate Node Visiting Frequency	222
7.10.	Untested Nodes on 001 SVA Program after 78,717 Tests.....	224
7.10.1.	Nodes Needing to Be Tested, But Not Tested	224
7.10.2.	Unnecessary Nodes	226
7.11.	Cut Extra Loop Repetitions	227
7.11.1.	Working Principle of “cut_iterations”	227
7.11.2.	Inputs and Outputs of “cut_iterations”	229
7.12.	Steps to Identify Unique Flowpaths.....	230
7.12.1.	The Steps.....	230
7.12.2.	An Example	231
7.13.	Count Visiting Frequency of Flowpath.....	233
7.13.1.	The Steps.....	233
7.13.2.	An Example	234
8.	References.....	238

List of Figures

Figure 2-1 Traditional “Water Fall” Software Development Cycle	21
Figure 2-2 Integrated Formal Software Development Approach	22
Figure 2-3 the 001 Tool Suite Structure	23
Figure 3-1 Horrible Loop.....	26
Figure 3-2 Test Cases Needed for Branch / Path Coverage.....	31
Figure 3-3 Calculate Total Number of Flowpaths	32
Figure 3-4 Nodal Coverage Marking Process1.....	38
Figure 3-5 Nodal Coverage Marking Process2.....	38
Figure 3-6 Illustration of Saturation Effect in Software Testing	47
Figure 3-7 Conversing of Intermediate Status Values into Input Values	50
Figure 3-8 Hazard Rate of J-M Model.....	57
Figure 3-9 Venn diagram with events A and E_1, E_2, \dots, E_n	92
Figure 3-10 Six Failure Observations Because of One Error	112
Figure 3-11 Tested and Untested Flowpaths, An Example	113
Figure 3-12 Tested Flowpaths, Part A	113
Figure 3-13 Untested Flowpaths, Part B.....	114
Figure 3-14 Ratio of Observed Reliability over Basic Reliability.....	117
Figure 3-15 Calculate Number of Flowpaths Theoretically	121
Figure 3-16 Identify Possible Flowpaths	125
Figure 3-17 Estimate Total Number of Flowpaths, Experimental Approach1	126
Figure 3-18 Predicted Total Flowpath Number \hat{N} Smaller than Tested Flowpath Number N_T	128
Figure 3-19 Illustration Ideal Number of Visits to	130
Figure 3-20 Illustration of Estimation of	131
Figure 4-1 SVA Demonstration Flowpath.....	135
Figure 4-2 Functional Map and TMap in 001.....	142
Figure 4-3 TMap Modeling the Concept of a “Table”	143
Figure 4-4 Primitive Control Structures in 001	145
Figure 4-5 Four arrangement settings for one SVA input data set	157

Figure 4-6 Cut Extra Iterations of a Looping Structure	167
Figure 4-7 Number of Visits to SVA Nodes after 198,321 Tests	170
Figure 4-8 SVA Unique Node Unreliability	172
Figure 4-9 Number of Test Cases vs. Number of Flowpaths, SVA	179
Figure 4-10 Visiting Frequency of Unique 001 SVA Flowpaths	180
Figure 4-11 Errors Detected in SVA 001 Program	181
Figure 4-12 U.S. Average Software Defect Potentials	186
Figure 4-13 Tested Flowpath Unreliability in 001 SVA Program, Average Value Approach	187
Figure 4-14 Flowpath vs. Test Case of 001 SVA Program	189
Figure 4-15 Number of Visits vs. Flowpath Index for 001 SVA	190
Figure 4-16 Unreliability of Untested Flowpaths by Average Approach	192
Figure 5-1 Uneven Flowpath Hitting Rates	197
Figure 5-2 Consistency of Flowpath Visiting Probability Estimation	201
Figure 5-3 Bigger Tail of Exponential Function Estimated from More Testing Data ...	201
Figure 7-1 Oracle SVA Program Initial Screen	211
Figure 7-2 Oracle SVA Program Output Screen	211
Figure 7-3 TMaps Storing Input Data for 001 SVA Program	215
Figure 7-4 A Static Unexpanded 001 FMap Containing Looping Structure	227
Figure 7-5 A Flowpath with Three Repetitions of a Looping Structure	228
Figure 7-6 Identify Unique Flowpaths Step by Step	231

List of Tables

Table 3-1 Terminology Common to Black Box Reliability Models	55
Table 3-2 Example for Observed Reliability Estimation.....	118
Table 3-3 Example for Basic Reliability Estimation	120
Table 3-3 Illustration of Calculation of	123
Table 4-1 Classes of Problems Raised in Mapping SVA Specification	148
Table 4-2 Random Sampling Input Data for SVA.....	159
Table 4-3 Table3.11 on [Jone91]	185
Table 4-4 Predict Flowpath Visiting Probability along the Testing Process	191
Table 4-5 Unreliability Data from Flowpath Coverage Based Approaches	193
Table 4-6 Unreliability Estimation Results for the 001 SVA Program	194
Table 7-1 SVA Error Details	221

1. Introduction

1.1. Background

The problem of demonstrating a high reliability for software has become more important as the need to use digital instrumentation and control (DI&C) technologies in nuclear power plants has grown. In safety critical systems, the use of computers provides many potential advantages. These advantages include more sophisticated safety algorithms, improved availability, easier maintenance, reduced installation costs, ease of modification and potential for reuse.

However, because of the critical nature of the applications, these advantages must be weighed against the problems of ensuring that the computer system can be trusted so that quality is adequately ensured. Much of the resistance to using digital technologies has arisen in regulatory organizations. For safety-related functions, the operative policy has been that available analog technologies are adequate, their weakness are well-understood and their contributions to nuclear power plant risks can be assessed using PRA techniques. Thus, the technological and commercial imperatives that have driven digital technologies into most realms of life and high technology applications have been absent in the case of nuclear power.

In the work prior to this thesis [Ouya95], a combination of modern software engineering methods was identified to design and implement high quality safety-related software. The stated techniques include design process discipline and feedback, formal methods; automated computer aided software engineering tools and automated code generation. The question left is how to assess the reliability of the software. Unlike that of the hardware, software failures are exclusively caused by design errors. The good news about this feature is that once an error is removed from a digital program, it is gone forever. There will not be any problems because of fatigue as are observed in hardware components. The bad news is that unlike mechanical equipment, we cannot test a piece of software, measure its quality, and thereby deduce the performance of another piece of

software. Also, we cannot increase the reliability of a software program by the redundancy.

Thus, the focus of the work in this thesis is to provide a standard, systematic, automated way to test the safety-critical software applied in the nuclear power industry and to measure its quality. The target users include both the designers of digital systems and the regulatory organizations. The method should be agreed upon by both parties and serves as an intermediate interface between the two.

1.2. Objectives

This work seeks to develop a method for the testing of safety-critical software¹ and for the systematic quantification of its reliability. The safety critical nature of the software under question mandates very demanding requirements in regard to the completeness of the testing strategy and the preciseness of the reliability quantification method. Yet, we need to remember that what we are seeking is an engineering tool that can help us overcome obstacles to the introduction of digital technologies into the nuclear power industry. This requires the method to be practical.

Thanks to the relative simplicity of the software used in nuclear power plants, we are able to leverage more toward completeness rather than feasibility of the resulting testing technique. Aware of the impossibility of a fully complete testing strategy even for an extremely simple piece of software, a systematic integrated testing method is developed. It incorporates Monte Carlo input data sampling (utilizing knowledge of the likely patterns of use of the software), software structure identification, nodal coverage checking and feasible flowpath checking. An experiment-based method is designed to quantitatively estimate the probability that an untested flowpath within a given piece of software is encountered during the next execution at any stage of the testing process.

¹ Regarding testing techniques, the emphasis of this thesis is the integrated testing. Though it has not been covered extensively in this thesis, it is very important to apply extensive, or even complete testing on the most important modules of the software before the final integration and finished product testing stage.

Closely related to the systematic integrated testing method is the second goal of the thesis which is a Bayesian updating method to estimate the software unreliability² — the probability of failure upon demand. A software failure is said to occur when the behavior of the software departs from the result obtained from an oracle (an independent mechanism to decide the correct behavior corresponding to the same input data set). Failures are the result of a software error or a design defect being activated by a certain input to the code during execution. Software unreliability estimation is performed at various stages during the process of testing software. These estimates are used to evaluate whether the software reliability requirements have been or might be met for the system as a whole. The estimation result provides both a measure of the software quality and also feedback to the software designer.

There are two activities related to software reliability analysis, estimation and prediction. For both activities, statistical inference techniques and reliability models are applied to failure data that is obtained from testing or during operation. Estimation is usually retrospective and is performed to determine achieved reliability from a point in the past to the present time. The prediction activity, on the other hand, parameterizes reliability models used for estimation and utilizes the available data to predict future reliability. In this work, both activities are carried out on the target software. An estimate is made of the unreliability of the part of the software that has been tested and a prediction is made of the unreliability of the part that has not been tested. Both are done with available data obtained from the tested part.

The reliability estimation method should be chosen so as to limit the estimation result uncertainty. The strength of the Bayesian updating method is its ability to incorporate both subjective and objective knowledge in the effort to estimate the parameters in question. It enables us to obtain the best estimation based on available testing information. Hence, it narrows the uncertainty.

There are various existing reliability growth models that have already been developed and implemented. The two major categories are black box models and white box models. One way to compare these available methods and to choose the one that

² Software reliability is defined as the probability of success upon next execution. Its value is equal to one minus the unreliability. Because both reliability and unreliability are measures of quality of the given software, they will be used interchangeably throughout this thesis.

best serves our purpose is to implement both and observe the results. But in this study, we need to do more than find a value for the reliability of the given software. We intend to learn as much about the software's internal structure as possible. A Bayesian technique has been chosen for this purpose. Currently all available Bayesian reliability growth models are black box models, i.e. the internal structure of the software is not examined. It is a more attractive idea to combine a Bayesian updating technique with a white box reliability growth model, by which quality-related features of the internal structure of the safety-critical software can be extracted. The most important factor or result of such a reliability estimation activity in this research is the same as in the PRA methods that are widely adopted in the nuclear industry. It is not the probability value that is obtained at the end of the reliability estimation process that is the most valuable to us. Instead, it is the relative reliability of the unique safety features of different parts of the software that is more significant.

Systematic and standard Bayesian based white box reliability estimation strategies are adopted in this study. They are intended to incorporate as much information as possible from the target software. The software features identified throughout the software quality assessment process provide a deeper understanding of the software itself. The uniformity of the method when applied to different safety-critical software should provide a standard measurement of the probability of success during a subsequent execution. A software quality standard can be generated from this standard process. The reliability value of a given piece of software estimated from the same procedure can be compared with external requirements and a decision made as to whether the software is ready to be deployed for unrestricted use.

The software quality assessment designed and demonstrated in this study does not demand the use of the formal engineering software development method discussed in the previous work. However, the use of the automated tools inherent in such development methods renders the required testing program much easier and more complete than other approaches.

Besides designing a complete and reliable testing and quality estimation strategy, the discussed techniques are implemented on a typical safety-critical piece of software that has moderate complexity among digital programs in the nuclear industry. This is

done to show the feasibility of all the stated approaches. The demonstration of the method on the sample software provides us an opportunity to consider a sound solution from both a scientific point of view and an engineering perspective.

1.3. Main Contributions

The work of this project can be divided into two parts, complete testing and reliability quantification. The methodology design and implementation on the sample software is carried out in parallel in order to guarantee the feasibility.

1.3.1. Grey Box Testing Technique

The testing technique that results from this study should be as efficient and complete as possible. In order to design the method, the literature was extensively reviewed, with completeness and general usefulness examined for each available technique. Generally speaking, there are only two major categories among the huge number of diverse types: black box and white box. 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 can be very useful because they simulate the actual usage of the program. The tester of the program does not need to examine the internal workings of the code. Exhaustive input testing is the only existing complete black box method, and it is very impractical. White box testing methods generate test data by examining the inner workings of the program in question. By correlating input data with the control structure, it is a more directed strategy and thus more efficient in error detection. The only complete testing technique within this category is the complete flowpath testing method, which also encounters practicality difficulty, especially when the software contains many looping structures. Under such circumstances, it is very hard to judge what are the possible flowpaths that can be accessed by some combination of

input data. Thus, it is impossible to calculate how many such flowpaths exist in the piece of software. Furthermore, the task of identifying certain input data that can activate certain flowpaths in the software is an easy one.

It was apparent from the literature search that none of the existing testing techniques was a good candidate to satisfy the completeness and feasibility requirement of this study. Our strategy was to design a method that combines strengths from both the black box and the white box testing techniques. Because it has the flavor of both testing categories, it is called the grey box testing method.

We start to generate input data by regarding the target software as a closed box. The specification and other related knowledge about the software is studied, the most likely patterns of use of the software are identified, and the operational environment is projected. A probability density distribution of the input data is then approximated from the study result. A Monte Carlo technique is applied to sample software input data randomly based on the probability density distribution function. Up to this stage, we are using a black box strategy to generate a large amount of input data in a very small amount of time. However this approach is not exhaustive according to the black box completion criteria. We feed the input data into the program. While testing the software and getting the testing results, the flowpaths activated are identified and recorded. Given that it is impossible to cover every flowpaths, we cut looping structures that include more than two iterations into loops that contain only two iterations. Because of this cut, our method is called a feasible flowpath coverage approach. After some tests are done, the nodal coverage situation is checked. It is not possible that all the flowpaths are checked unless every node in the software has been checked. As shown in our demonstration example, the chance of identifying unnecessary nodes and of detecting hard-to-find errors in the software is very high when efforts are made to check any uncovered nodes after a fair amount of tests. We do not know the number of possible flowpaths in the software. Hence the probability of encountering an un-identified flowpath can always be estimated by an experimental method based on the flowpath information that has been accumulated. This later flowpath and node identification part is more of a white box testing strategy in that we are aware of the internal structure of the software and are trying to take advantage of the knowledge obtained during testing. Because of the utilization of both black box

and white box testing techniques in this study, we name the testing method used in this study as grey box testing, which is a mixture of both available approaches.

1.3.2. Nodal Coverage Based Bayesian Reliability Estimation Methodology

The reliability of a software program is defined in nuclear industry as the probability of success upon its next execution.

Existing software reliability models can be classified as either black box or white box groups. Black box reliability models consider only failure data, or metrics that are gathered if testing data are not available. They do not consider the internal structure of the software in reliability estimation and are called such because they consider software as a monolithic entity, a black box. For software built from reused or commercial off-the-shelf components, architecture-based or white box reliability models are developed. In these models, components and modules are identified, not in a traditional software engineering architecture sense but rather in the sense of interactions between components. The interactions are defined as control transfers, essentially implying that the architecture is a control-flow graph where the nodes of the graph represent modules and its transitions represent transfer of control between the modules. Failure behaviors are combined with the architecture to estimate overall software reliability as a function of component reliabilities.

Based on the features of nuclear safety-critical software and the strengths of both available categories of software reliability models, an intermediate model was developed during our research. From the black box group, we were especially interested in the Bayesian model. It considers both the number of faults that have been detected and failure-free operations. Detection and perfect removal of errors in the software increases the reliability of the software. Failure-free tests on the software improve our confidence in the software. Furthermore, the very unique advantage of this Bayesian model is its ability to take systematically any subjective information into consideration. This is the reason why we want to adopt the Bayesian model. From the white box group, on the other hand, we have learned that it will be very helpful if we can incorporate structural

information about the software into a reliability estimation model. To do so can help the tester obtain deeper insight about the different importance of different parts of the software and their individual confidence levels. Again, because the target software that we are studying is safety-critical, the more knowledge we have about it, the better. However, our approach in this study is unlike the available white box models in that we do not divide the software into smaller modules because the software itself is small enough. Rather, we go back and use the traditional software architecture concept. The component of the software is the traditional node. There are no interface and control transfer concepts involved in our model.

In this approach, we adopt a white box strategy to look at the reliability of the software, incorporating both failure and structural information into the model. Each component of the software (that is each node) is examined and its reliability is estimated using the Bayesian method. It is required that all the nodes in the software be tested.

1.3.3. Flowpath Coverage Based Bayesian Reliability Estimation Methodology

A further step from the nodal based reliability estimation approach is a flowpath coverage-based reliability model. But again, even for very simple software, the requirement of testing every flowpath is too hard to complete. There is also no theoretical method to calculate how many flowpaths exist within a program because of the impossibility of accessing some of the theoretically existing flowpaths. The first step to estimate reliability of the software in this approach is the same as that described in the proceeding section, i.e., using a Bayesian method to estimate reliability of every tested flowpath. After examining the different errors encountered in the projects, we initiated the concepts of type I and type II errors. If an error is of type I, it causes a failure every time its host flowpath is visited. If an error is of type II, it has the same probability to cause a failure during any visit to its host flowpath. It is apparent that because of the existence of type II errors, we can never claim one hundred percent reliability on a software system because there is always a tiny probability of a type II error not being detected even if a large number of tests have been done on a flowpath. Because we are

not able to achieve a complete flowpath coverage testing, we first estimate the overall reliability of the flowpaths that have been tested. In order to incorporate the reliability of the untested flowpaths in the model, we have to come up with two estimates: the probability that an untested flowpath is visited upon next execution of the software and the reliability value of the untested flowpaths. The first value is approximated by an experimental approach and the second is estimated using information from the tested flowpaths in the light of the many commonalities shared by those two parts.

1.4. Thesis Organization

A brief summary of the work done prior to this thesis is presented in chapter 2 where other stages of the formal software development cycle were achieved. The whole picture is given and the position of the work done in this thesis research is pointed out. In chapter 3, the grey box testing strategy and Bayesian based white box reliability estimation techniques are discussed in full detail. Both testing and software reliability models from the literature are studied. The assumptions, data requirements, action steps and formulas are described. In chapter 4, all the methodologies are implemented and demonstrated on the sample nuclear safety-critical software. The experiment details as well as reliability estimating results are reported. Chapter 5 provides some discussion on the demonstration results. In chapter 6, I conclude this thesis with a summary of the major findings of this work. Possible future work is described.

2. The Integrated Formal Approach — DBTF (Develop Before The Fact)

The objective of this study is to improve the quality of the safety-critical software used in the nuclear industry. Two steps, one in the software developing stage and one in the software checking & evaluation stage, are taken to achieve it. In this thesis, the major effort is concentrated on the second stage while the first step is addressed by the selection of a formal software development methodology, DBTF and its supporting tool, OO1 CASE Tool Suite. For completeness of this thesis, some major points about DBTF are mentioned briefly in this chapter. Some results of applying this approach to the sample software, SVA, which is studied intensively in this thesis, are presented afterwards. For more details about DBTF, including the criteria and reasons to select this software development method, refer to Ouyang's study report [Ouya95].

2.1. Integrated Formal Approach

A variety of integrated formal software development approaches have been studied. The target software system for this research is instrumentation and control software used in nuclear power plants. The literature associated with this type of software supports that a Develop Before The Fact (DBTF) method be chosen in this project.

To make use of the underlying software safe, scientifically-based formal techniques are introduced to replace the previous ad hoc approaches. The conventional software development process can be described in terms of a “waterfall” model as shown in Figure 2-1. It is generally agreed that formal software development should include more elements as shown in Figure 2-2, incorporating strong feedback between different elements of the design process. Academic research and industrial practice have greatly advanced software engineering to a more disciplined level where increasing amounts of mathematical formality are introduced. The term, formal method, is used to describe such a cluster of mathematically formal techniques. Formal methods aim to improve the

quality of software in two related ways: by providing a specification that is clear and unambiguous, using easy to validate mathematical statements of the required software behavior and by making verification during the software production process more effective and easier to audit. Formal techniques are used to address activities in three major blocks: specification, validation, and verification. Given that the formal specification and validation issues have been addressed in full detail in prior work (as a result, DBTF was chosen according to a series of criteria), the emphasis in this thesis is to answer the question of how to apply formal methods in the verification block, including systematic testing and structure-based reliability estimation.

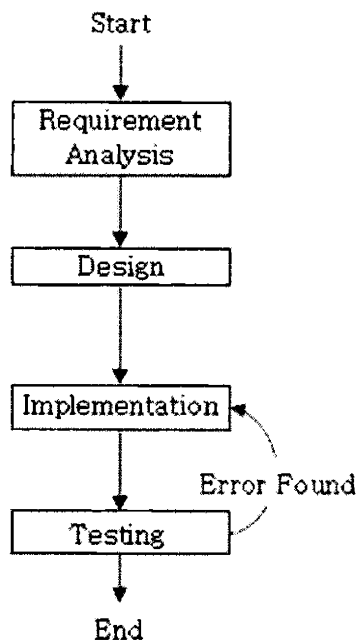


Figure 2-1 Traditional "Water Fall" Software Development Cycle

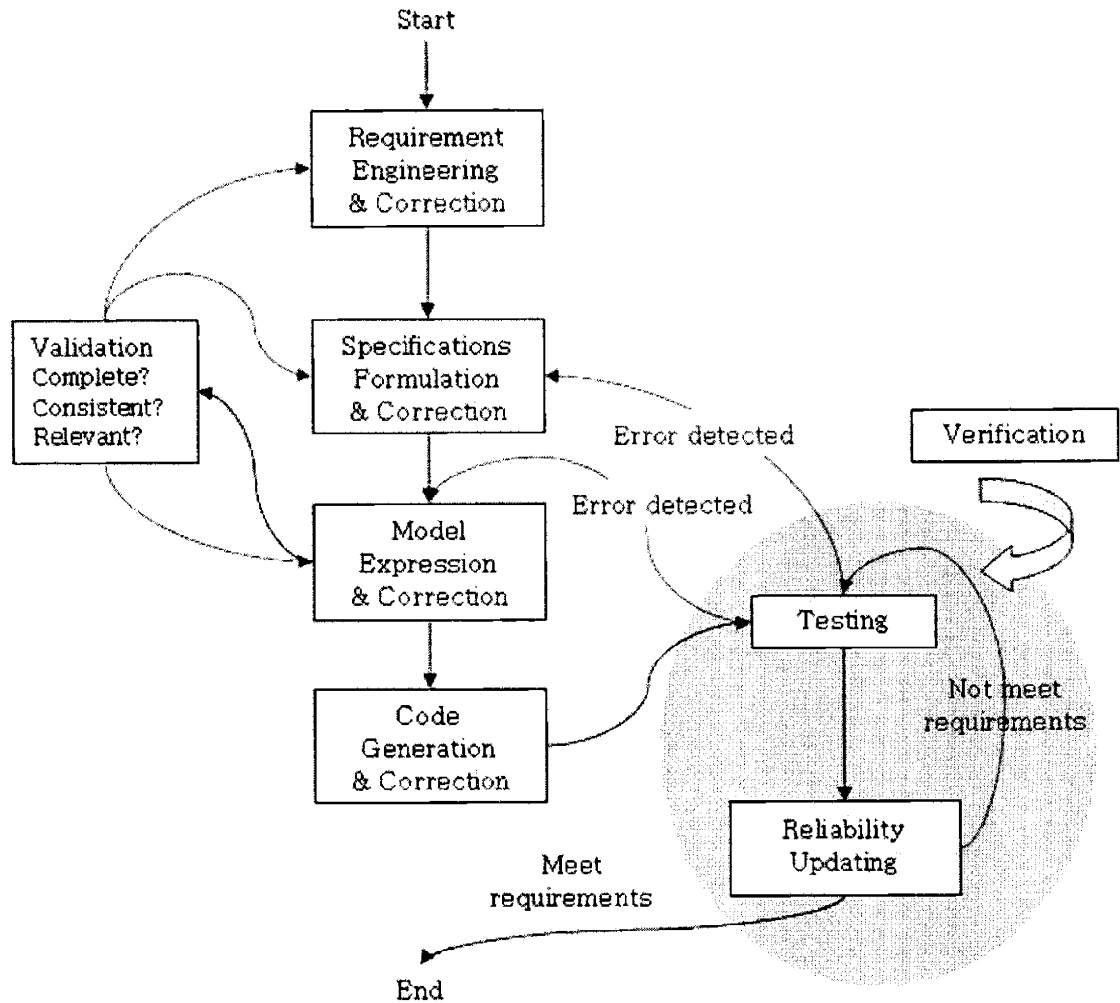


Figure 2-2 Integrated Formal Software Development Approach

2.2. DBTF Process

Development Before The Fact (DBTF) is a trademark methodology of Hamilton Technology. With DBTF, each system is defined with properties that control its own design and development. With this paradigm, a life cycle inherently produces reusable systems, realized in terms of automation. An emphasis is placed on defining things right in the first place. Problems are prevented before they happen.

DBTF is automated by 001 Tool Suite, an integrated system and software development environment. It can be used to define, analyze, and automatically generate complete and integrated production-ready code for any kind or size of software

application with a significantly lower error rate and a significantly higher reusability than with traditional approaches. The conceptual structure of 001 is shown in Figure 2-3.

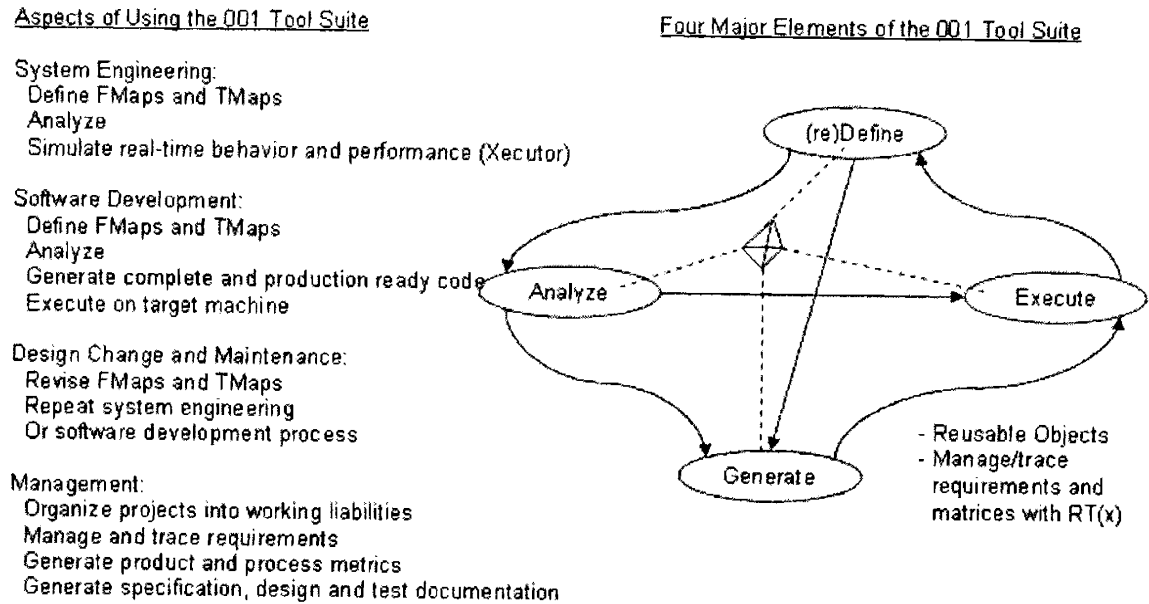


Figure 2-3 the 001 Tool Suite Structure

3. Integrated Testing and Reliability Estimation Strategies

3.1. General Assumptions

There are some limitations on the software systems, to which the testing and reliability methodology discussed in this thesis, can be applied to.

The first one is that the software should be relatively simple because it needs to be tested as complete as possible and use as much as possible information to quantify its reliability. Intuitively, the completeness and feasibility are always opposite to each other. At the first glance, this requirement seems to be very stringent, limiting the method development dramatically. But if we concentrate on the instrumentation and control software used in nuclear power industry, it is not the case any more. Software systems can be categorized into three groups according to the reliability, R (defined as the probability of error-free service, over life): fully testable ($R=1$), substantially testable ($R<=1$), and largely untestable (R unknown). For nuclear software applications, many can be made much simpler (i.e. entering category 1 and 2) than has been typical in most non-nuclear applications (category 3). This thereby removes large feasibility concerns about the testing and reliability estimation addressed in this thesis. In this study, we only consider software reliability estimation in categories 1 and 2.

The second assumption about the target software is that it behaves uniformly as time moves on. To be more specific, for the same input or inputs, the software always gives the same output whenever it is executed. If this is true, every unique set of inputs uniquely identifies a unique execution of the target software. The number of different input data sets is the number of different test cases. Every set of output can be compared to the corresponding oracle in correspondence to the same input data set at any time. If there is any error detected, the same failure case can be repeated by inserting the same input data into the software under testing. If we say the first assumption is made due to both testing and reliability estimation requirement, this assumption is almost purely for the purpose of achieving a reasonable testing solution. Again, this assumption is not very

much a limitation to the instrumentation and control software applied in nuclear power plants.

3.2. Definitions

3.2.1. The Control Flowgraph

Understanding of the control flowgraph is essential to understanding much of the literature on software testing. In general, the control flowgraph is a graphical representation of the control structure of a program. There are only two types of components to the flowgraph: circles and lines. A circle is called a node and a line a link. A link is a connection between nodes. A link from node j to node k is given the notation (j, k) . Node j is designated as the predecessor of node k , and node k the successor to node j . There can be only one link between any two distinct nodes. The first node of a program is known as the start node, and has no predecessors while the last node is known as the exit node and has no successors. A node with more than one link entering is a junction and one with more than one leaving is a decision. A node is either a decision node, or a successor of a decision node, or the exit node. It is important to notice the difference between a node and a program statement. A node in the control flowgraph may quite possibly contain more than one statement if it is not a decision node. The reason is that the flowgraph only cares about the decision making points in the program, thus highly simplifying the program structure by merging contiguous processing statements into one node.

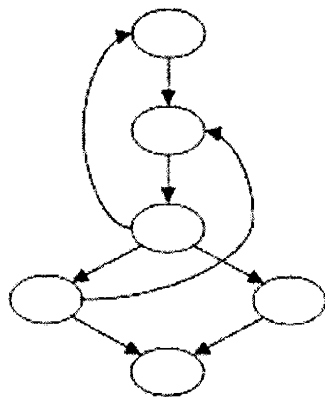
3.2.2. Flowpath

A path or flowpath in a program is a finite sequence of nodes connected by links. A path is designated by a sequence of nodes. A complete path is a path with the start node as its first node and the exit node as its last node. A loop free path is a path with none of its nodes repeated.

3.2.3. Feasible Flowpaths

In order to ensure complete path coverage, it is necessary to expand the program flowgraph so that every distinct execution path is stated explicitly, and then every such path is tested. In the event of looping structures, this expansion of the flowgraph structure may be unfeasible because of the complexity of the graph. In such a case, only limited testing may be practical. For this situation, the concept of feasible flowpath coverage is used. A feasible flowpath is a flowpath that contains at most k iterations of every loop in a given program or flowgraph. We have found that this approach will reveal many of the errors in a program, but it does not guarantee finding all [Fran88] [Ntaf88].

In our study, we have set k to two. For loops either with a fixed number, greater than two, or with a variable number of iterations, we only treat zero, one and two iterations of a loop as unique expansions of the flowgraph. If a loop is expanded to more than two iterations during an execution, all the repetitions numbered greater than two are removed from the flowpath record before any further analysis is performed on this flowpath. Also we assume there is no horrible loop in the program under testing. See the following pictures for feasible flowpaths with k equal to two and horrible loops.



Horrible Loop: "...code that jumps into and out of loops, intersecting loops, hidden loops and cross-connected loops..." by Beizer.

Figure 3-1 Horrible Loop

3.3. Testing Methodology

3.3.1. Testing Strategy Overview

In this project, our main purpose is to improve and justify instrumentation and control software reliability so that it could be introduced into nuclear power plants. Safety has always been a critical issue in this industry. The potential for injuries and loss of property makes it imperative to test the digital systems extensively and to estimate their reliability accurately prior to the deployment. As a result, the unreliability requirement typically cited for safety-critical software is extremely high, with typical failure frequencies of about 10^{-7} per hour. It has been suggested that this is an unrealistic requirement given the enormous amount of testing necessary to assure compliance with this level of reliability [Litt92]. Also, it might be argued that, from a technical point of view, the software reliability is only guaranteed if it is operated within a specified environment. Taking the above factors into consideration, a compromised and balanced solution should be provided. Both reliability and practicality are essential in this problem.

3.3.2. 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 our research is to find a new testing technique that is proper for safety-critical software used in Nuclear Power Plants. The resulting method should be as complete as possible because of the safety-related feature of the software in question while kept practical enough for real life use. In order to achieve this goal, the literature has been extensively reviewed, and each method has been rated for both completeness and general usefulness. In the end, a new testing technique is introduced that incorporates features from the two major existing testing methods, giving the

maximum degree of completeness while maintaining the testing workload at a reasonable level.

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 here because of their lack of generality. In this study, the final chosen method should be useful in general as well as in specific cases. Many of the methods found fit this criterion. Although the methods are quite diverse, they can be divided 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.

3.3.2.1. Black Box Method

Black box testing methods, also called functional testing methods, derive test cases from the software 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. For example, a simple program's specification document might indicate that two numbers (A and B) are taken from user input and their sum (A+B) is displayed on the screen. To test this simple piece of code functionally, an input set is chosen that represents the typical use of the code. The tester may wish to input several combinations of A and B. For instance the cases where A is equal to B, A is greater than B, and A is less than B could constitute a test set. The tester may wish to use input data values, which are more likely to cause errors. For instance, letters of the alphabet could be entered instead of numbers. If all of these cases execute properly, the code is found to be satisfactorily tested functionally. However, if one of the functional test cases yields an incorrect result, the code must be examined for the error. Because of the nature of this strategy, black box testing is also known as input / output testing.

Recall that the focus of this search is to locate a complete, yet practical testing method. The only complete black box method is known as exhaustive input testing. For

the example above, every possible combination of the two numbers would be tested, resulting in an infinitely large test set. Exhaustive input testing is very impractical. If any one of the input variables is a rational number, an infinite number of test cases must be considered. Despite the apparent impracticality of complete black box testing, these methods should be mentioned. They are examined here as possible complements to another more useful testing method. It may be possible to use them in conjunction with other testing methods to achieve a more complete result. Also, this approach, which is common across many engineering disciplines, has several significant advantages. The most important advantage is that the testing procedure is not adversely influenced by the component being tested. For example, suppose the author of a program has made the implicitly invalid assumption that the program would never be called by a certain class of inputs. Acting upon this assumption, the author might fail to include any code dealing with that class. If test data were generated by examining the program, one might easily be misled into generating data based upon the invalid assumption. A second advantage of black box testing is that it is robust with respect to changes in implementation. Black box test data need not be changed even when major changes are made to the program being tested. Another advantage of black box testing is that it simulates the actual usage of the program. A final advantage is that people unfamiliar with the internals of the program being tested can interpret the result of a test.

The Black box methods will always be useful in testing small pieces of code designed for simple and specific tasks. Exhaustive input testing is the only known complete black box testing method, and is very impractical. But because of the above mentioned advantages of this method, it is very promising that black box methods can be used in parallel with other testing methods to achieve an effective and close to complete testing result.

3.3.2.2. White 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 because the basis of the method is 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 also necessary to 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. Only some basic methods are reviewed in this section. In reality, as in the black box testing group, none of the techniques satisfies both the feasibility and completeness requirements at the same time. As discussed later, even complete flowpath testing, which was believed complete before, is not necessarily perfect in terms of detecting errors, let alone feasibility. The goal here, again, is to learn some useful lessons from the existing white box testing techniques and use them in conjunction with other methods to improve software reliability.

There are three basic structural testing techniques. They are statement, branch and flowpath testing methods. These form the basis of understanding for most other forms of structural testing. These testing techniques are said to result in some coverage of the code.

Statement coverage is a testing method, which assures that every statement of the code has been tested. In other words, the test set selected results in the execution of every statement in the code. Statement coverage is the least rigorous structural testing method. It is often possible to achieve statement coverage with relatively few test cases, even for a large piece of code. At the end of the test campaign, the tester knows that all of the statements in the code are executable, and that the program has yielded some correct results. However, the program tester cannot conclude that the code is error free because this method is insensitive to some logical control structures. Complete statement coverage is a far too limited testing criterion to be used to indicate successfully complete testing.

Branch coverage is achieved by testing every transfer of control from one program statement to the next. To attain complete branch coverage, every direction at every decision node must be executed at least once by test cases. In the context of control flowgraph/node/flowpath, branch coverage is equivalent to nodal coverage. Each successor of a decision node (we will call them alternative nodes from now on) represents

a unique direction taken at that decision node. Path coverage involves testing every combination of every direction at every decision node of a program. Compared with branch coverage, path coverage is the more rigorous method. In fact, path coverage is said to be the only complete testing method in the white box group. However, even complete path coverage does not ensure detection of all errors as some may be found only upon subsequent tests of the same path.

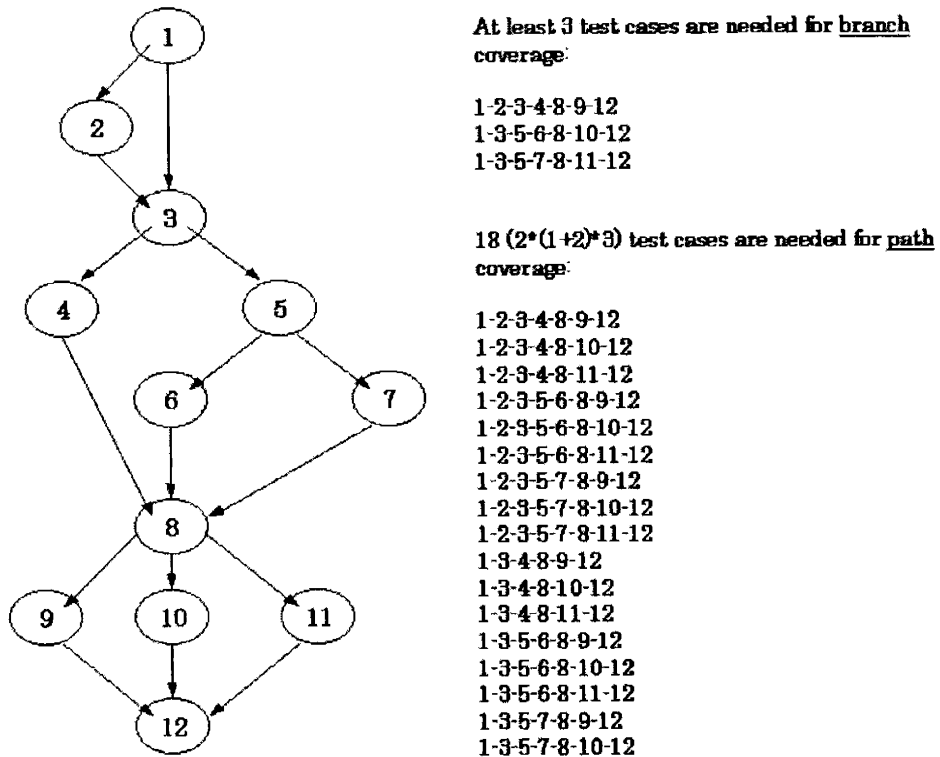


Figure 3-2 Test Cases Needed for Branch / Path Coverage

Let us assume for now that the path coverage is complete. A potential problem with such a testing strategy is that there are often too many different paths through a program to make it practical. The number of paths is exponential to the number of branches. Consider a simple program with a 100-iteration loop. There is an if-then-else decision structure inside the loop. The if-statement causes either the true or false branch to be taken, and both of these paths go on to the next iteration of the loop. Thus, for each path entering the i th iteration, there are two paths entering the $(i+1)$ st iteration. Because there is one path entering the first iteration, the number of paths leaving the i th iteration is

2ⁱ. Therefore there are 2¹⁰⁰ paths leaving the 100th iteration. Testing each of the 2¹⁰⁰ paths is not likely to be practical.

The second problem is that many paths are impossible to reach due to the relationships of data. Look at Figure 3-3 for a simple example.

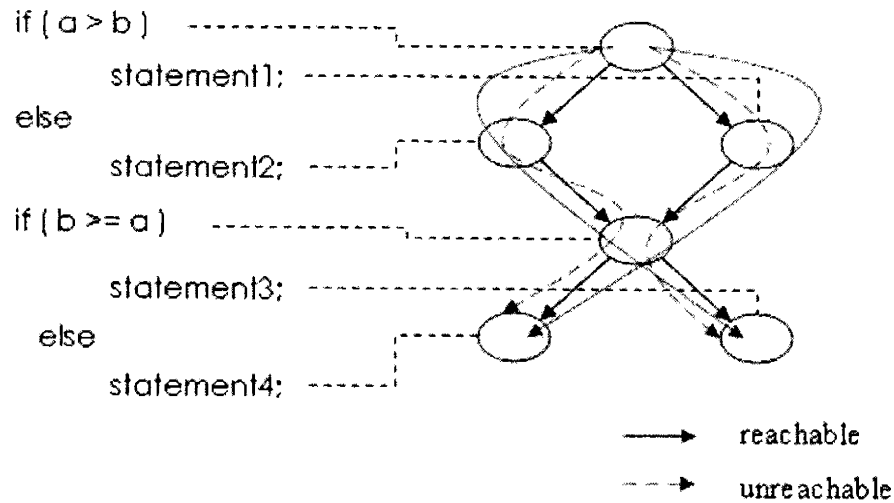


Figure 3-3 Calculate Total Number of Flowpaths

Assume neither of statement1 and statement2 change value of a or b

Path coverage considers this fragment to contain four paths. In fact, only two are feasible: success = true or success = false. This raises another question for: what is the total number of reachable paths in a program.

Another problem is that a testing strategy based on exercising all paths through a program is not likely to reveal the existence of missing paths, and omission of a path is a fairly common programming error. Test data based solely upon analysis of the code structure is not going to be sufficient. One must always take the software specification into consideration.

A final problem with path coverage, found during our demonstration work, is that even if a path has been tested, we are not 100% sure that the path is free of error. Without taking into account the missing paths, still, a program can never be tested

completely in terms of detecting errors. The reason is that there is always a non-trivial probability that errors might exist in a path that has been tested a finite number of times.

3.3.2.3. Literature Conclusion

The goal of this literature review is to locate as close to a complete method as possible, with a focus on feasibility at the same time. Also, only those methods that are applicable in general have been studied.

There are two major groups of testing methods: black box testing and white box testing. Each method has its advantages and disadvantages. Each group has one complete testing method. Among black box methods, the exhaustive input testing is known as complete. But implementation of exhaustive input testing results in an unreasonable number of test cases. Among the 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. Also, the literature regards complete path testing as being impossible for all but the simplest loop free programs. The presence of a loop may result in an infinite number of test paths, resulting in an infinite testing time. 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 to 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 a 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 that we have developed and adopted in this study is discussed. From the name we give to our testing method, “gray box” method, it is very easy to imagine that it is a technique standing between white box and black box.

We are trying to incorporate merits from both techniques and also circumvent their impracticalities.

Because the testing process is very complicated, there are many details that need to be given careful thought in order to make a theoretical method practical. Among them, how to generate the input data, how to determine the correct answer for each test case (the correct answer data base is known as an oracle), how to automate the answer checking process, how to record the testing results, etc. These issues are covered in more detail in the following sections.

3.3.3. 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 natural connection, resulting in an approach that is both practical and satisfactory for our high reliability requirements.

3.3.3.1. Method Overview

The testing technique literature review tells us that there is no method at hand that fulfills the requirements of this study. We intend to design a testing technique that combines the advantages of current available methods. We use the black box method to generate input data because it can simulate the actual usage of the software in question given that the specification document is always available for testing purposes. 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. Also it is an automated process by nature. The flowpath coverage testing technique is inverted so that it can be used in combination with the black box input data sampling process and its working speed is increased dramatically. We look at the program itself as the software control flowgraph and assume that 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. A rough reliability estimate of the

software could be obtained at this point. After this is achieved, we continue to test the program more thoroughly and through a reliability estimation technique that is discussed later 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 that 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 section 3.3.3.2, the testing process as a whole is presented more extensively. More coverage on all the particular issues concerning testing is given in section 3.3.4.

3.3.3.2. 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 [Sui98]. The formal process includes five steps. It takes in a program and through thorough analysis produces test cases systematically, which achieve structured path testing coverage involving no infeasible test cases. In brief, the five steps [Lyu96] 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 the 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 in 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 because of the data relationships.

In order to solve the above dilemma, we decided to adopt the trial and check method. It is well understood how to generate input data without examining the source code both manually or in an automatic manner. We argue that because of the safety-critical feature of the software under our study, we should carefully design a set of input

data from the software specification. Manual input identification may be applied during this stage. This is very important first step for it enables us to earn confidence about the software under well-known operating scenarios. Secondly, with some operating profile of the input data being used, the Monte Carlo method could be used to computerize the process of generating input data. The 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. The input variables to the Monte Carlo method are random variables with known probability density distributions. In each single trial of the Monte Carlo method, the values of input variables are generated randomly using the probability density distributions, the system responses to the generated input variables are computed, and the outcomes recorded. There are several benefits to this random sampling process. First, the obstacle to automating the process is inherently removed. 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 associated probability distribution is of a good precision. Finally, there is one last method for the input data production. That is to get it directly from the user. Unlike other software input data, no input can be obtained without the source code. For example, many process control or data processing programs used in nuclear power plants take input data directly from sensor readings or other time existing well-known plant indicators. If a large number of those real data could be obtained before the software is actually applied, that would 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 it comes to the check portion. The target program is fed with the generated input data and the paths executed are identified and recorded. Our first step in the check process is to achieve complete branch coverage. The actual implementation is very simple. We take the program as the flowgraph. Each execution of the target program leads to coverage of a sequence of nodes — the executed path. For each tested path, all the nodes on the path are marked on the flowgraph. Each node is associated with a counter. Before any test is done, all counters are set to zero. Every time a node is found within a flowpath, the counter of that node is increased by one. The value of a counter remains zero, if no visit

has been made to its associated node. Otherwise, the counter stores the number of visits that have been made to its corresponding node. After an amount of tests are performed 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 transferred to a file. If no modification has been made throughout 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 previous node list files. Because 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 previously 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 3-4 and Figure 3-5 for a visualized description of the marking process. The marking job is done after a specified number 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 that will lead to the flowpaths that bypass those unvisited nodes should be performed. It is possible that some surplus nodes that are impossible to visit will 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, a higher error rate will probably show up from the last few hard-to-visit nodes. The reason is that usually these nodes are dealing with rare events, 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 can be estimated from the visiting frequency during testing for each node and number of errors found on each node. This will be discussed later in the reliability estimation section. When dealing with a program that belongs to the testable group (refer to section 3.1 for software categories according to their testability), such as the one used in our demonstration work, it is often the case that some useless nodes are found when trying to identify input data for every branch, or node, in the software. This is a beneficial result because those unreachable nodes only add confusion to the program.

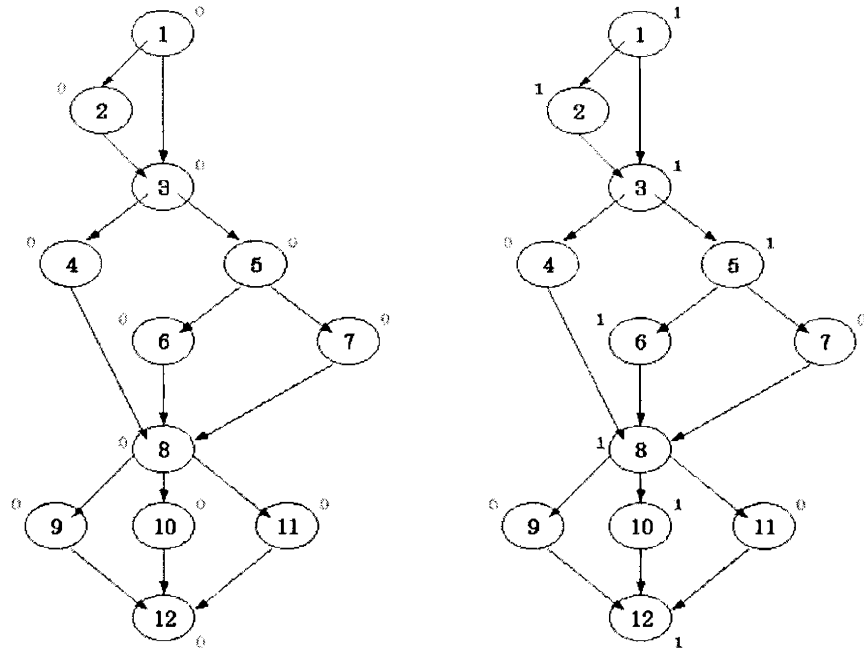


Figure 3-4 Nodal Coverage Marking Process1

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

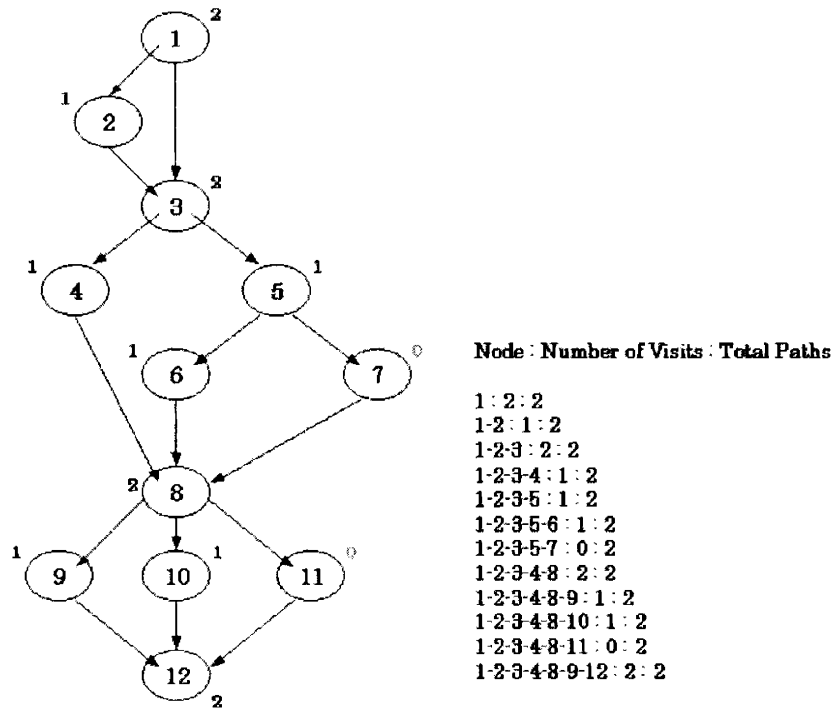


Figure 3-5 Nodal Coverage Marking Process2

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

After complete nodal coverage has been achieved on the program under examination, the same testing process may or may not be continued depending on the reliability requirement. It should be noted that reliability estimation can be made at any testing stage, regardless of whether complete nodal coverage has been achieved. During the testing process, the software reliability based upon path reliability is estimated at specified 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 a later section. But there is one piece of very important information that is mentioned here. When we estimate the reliability, we do not simply assume that every flowpath has the same importance. Because we assume that the program is tested according to the actual operating profile, or according to a very close approximation, we put more weight on the more frequently visited flowpaths during the testing process. This is because we are not blindly seeking complete path coverage. Rather, it is only treated as one of several indicators that show how complete the testing process is. We can get an idea of how many more test cases (sets of input data) we need in order to test a specified fraction of the entire set of flowpaths. If some reasonable reliability level has been achieved and a large number of new input data are needed in order to cover only a few untested flowpaths that are rarely visited in reality, we may decide to stop the testing process because the remaining large effort having to be made in order to achieve complete path coverage is not worthwhile.

3.3.3.3. Conclusion

In this section, we discuss the framework of the gray box testing technique, used in this project. In the following sections, the task of testing safety-critical software is addressed from several perspectives, determining the operational profile, generating test cases, making 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 to

testing software can be overcome via advantages in technology. Some that are more fundamental in nature remain unsolved.

3.3.4. Testing Details

3.3.4.1. Operational Distribution

In order to make the testing process more efficient and the reliability estimation more accurate, obtaining a distribution of input data during actual operation is always a necessity. But, in general, for a process control system, the only probabilities that can be reasonably estimated are those of the occurrences of various events in the environment. These probabilities must then be mapped to the distribution of inputs that the program will encounter during actual operation. For some generic control algorithm, even when the distribution of inputs and states can be determined for some parts of the input domain, it is impossible to be complete or nearly so because we are not even able to know what are the related events. Another complex contributor is the fact that, in many control systems, a human is involved in the operating loop, whose decisions can change the whole operation profile dramatically.

In light of the importance of obtaining a good input domain distribution as well as the difficulties involved, we suggest several partial solutions to get as much solid information as possible while maintaining a manageable workload. First, we look at the program design document. Most of the traditionally designed digital control software specifications are expressed using pseudo programming languages. The purpose of a specification is to describe every procedure scenario that the designer could possibly acquire. This information is exactly what we need to obtain some operational profiles. We can never depend upon the specification to obtain a precise and complete operational input distribution for two reasons. A specification is relatively abstract. It is more about what tasks the program should complete than how the tasks should be completed. Many details are still waiting for the software developer to address. If we were to partition the input domain based solely upon the specification, many minute, subtle, but important points in terms of testing might be ignored. Another reason is that human work instead

of a computer calculation is necessary in order to find the operational profiles through this method. This makes it almost impossible to be complete when translating the software specification into an operational input distribution. In spite of the above weakness, we are still able to claim that to obtain partial operational scenarios from program specification is very useful, handy and efficient.

If the specification itself is too abstract, why don't we look at the more detailed part, that is, the program itself? It certainly contains some of the details that we need to get the operational profile. Thus, analysis of the program structure can serve as the basis of a complementary method. The program itself is more detailed and more about what the programmer does to achieve every function mentioned in the specification document. But, in order to take advantage of the additional information contained in the program structure, even more human effort is necessary. Moreover, such information is even further away from the real operational scenarios during field use. Thus, we argue that this method could only be used as a supplement.

The only way to get very precise operational profiles is to put the software into real field use, or get a large number of real input data to test the software. A nearly equivalent way of doing this is to develop a full-fledged environment simulator, which can be complex, error prone, and also require lots of testing. For most situations, it is very hard to use the software on the spot before little testing has been done. To build a simulator is expensive and time consuming compared with the development and testing work for the software to be tested, although an environment simulator has an extra advantage in that it simplifies the determination of the correctness of the output.

In this project, we only implemented this first method while taking the second into consideration to some extent. We did not have the opportunity to obtain a large real input data sample from our industrial partner nor did we have the time needed to develop a simulator. More details are provided in the demonstration section of this project.

3.3.4.2. Input Data Generation

There are two major techniques used to generate test data. The first one is to sample independent, identical input data points according to the operational distribution.

The second one is to randomly generate uncorrelated test data within the input domain. When implementing this method, the testing process should be instrumented to convert status variables into input variables (save values of status variables from the previous execution and use them as part of the input data to the program upon current execution), which renders the program being tested not exactly identical to the actual program. Both of these two methods have their advantages as well as disadvantages. On one hand, because of the statistical nature of our reliability estimation method, we have to get enough testing data from tests to permit a useful distribution. On the other hand, due to safety-criticality of the software that we are testing, we have to make sure all the important logical scenarios covered in the design documents are tested to be correct.

In this work, we used both of the above these techniques to generate testing input data. For the first method, in the beginning, we obtained as many as possible input scenarios either from the design specification or from analysis of the program structure. A complete algorithm requirement design should be completed at the time the software specification process is initiated. For this reason, a majority of the program scenarios during field use should be included in the design document. The given system may spend most of its time in those well-understood operating regimes. As a result, checking all the scenarios covered in the specification is the most convenient and fastest method to test the software in the first place. There is an important point that needs to be emphasized. That is, test cases should be chosen in such a way so that while the middle range of the logics should be checked, the boundary situations are even more important.

Unlike random input data sampling, it is very hard to automate the generation of a series of correlated test data in accordance with some of the operational scenarios. Test scenarios are constructed manually, which precludes automated generation of a large number of such data, which then prevents obtaining statistically significant test results for the statistical reliability estimation. The importance of this method combined with the difficulty in automating it leads us to apply only a small number of input data to the target program in this fashion.

The second method to generate test data is to randomly sample the input data domain. This method is similar to the technique used to determine hardware reliability, selecting a random sample of components and subjecting them to operational use. It is

fairly easy to make this process automatic. Rather than the detailed input distribution, only some basic information, for example, the legal range of inputs and the dependency among inputs within the same input data is needed. The input data generated in this manner could cover a lot more scenarios than the first method. Some “unexpected” faulty points in the program might be found. The automation also makes it possible to achieve a much larger sample of test data, which, hence, makes the reliability estimation more accurate and more statistically significant. Still there is another problem that needs to be address. That is, a process control program, such as what we use in our demonstration, usually has internal states. These internal states are hard to estimate or generate randomly. They also make the tests persistently dependent on one another for some period of time. We have no choice but to turn these internal states into some form of input, thus making them controllable to some extent. One other minor issue concerning input data is the possible human involvement in the program running loop. Some simplification must be made in order to skip the human-computer interaction thereby making automation of the testing process possible. Random sampling is used again to simulate the operator controlling commands, though it may not match the actual practice. As throughout our whole testing and reliability estimation process, we make every possible effort to obtain a good compromise between accuracy and feasibility.

3.3.4.3. Verification

Verification is a very important modular testing method. Formal verification can be used for proving the correctness of lower level components, general properties of upper level components, and the correctness of the system under normal operating situations. We argue that the higher the reliability achieved by a system before testing, the fewer tests are needed in order to achieve specified reliability value. A hierarchical design structure, like the 001 AXES language used in our project, organizes the software into a series of abstraction layers in which a component at any given level only uses components at lower levels. Doing this improves the testability and provability of the software because the debugging and verification can proceed level-by-level starting with the lowest one. It is true that we cannot solely rely on program verification to trap all the

faulty points. However, it is certainly of great assistance in reducing the workload at the integrated testing stage. It should be emphasized that the proofs obtained during the verification stage do not directly reduce the number of test cases. The only way of directly reducing that number is to prove complete path correctness. However, we can indirectly reduce the integrated testing effort when a specified reliability objective is required. Because verification can substantially improve confidence in the correctness of the program, we can express such a belief in the form of the prior distribution of the program reliability estimation value. General knowledge about the software development and verification tool is required in order for proper prior parameters to be identified and used. The more information we know about the verification process and put into the prior distribution, the quicker the integrated testing process will lead us to attain the real software reliability estimation and the fewer number of test cases will be needed.

3.3.4.4. Output Determination

In order to complete testing, an oracle is always required to determine whether the output of a given input data set is correct. For a small number of input data used to test a small program, manual checks can serve as a reasonable source for the oracle. It is especially convenient if we want to test the scenarios described in the software specification. The correct outputs corresponding to the inputs that lead the program to go through that carefully studied scenario is known from the specification. However, for the large number of test data that is required by the sampling model, it is essential to automate the test oracle. But, without a full-fledged simulator, to achieve complete checking via automation is a very difficult task. In practice, a combination of partial solutions employing automation is usually used. These include:

- a) Embedded assertions, which are conditions inserted into the program at various places. If false, warning messages are displayed signaling the possibility of faulty output.
- b) Reverse checks, which is to substitute the program result as the input.
- c) Reasonable checks, checking if the output values are plausible within given context.

- d) Interpolation checks, that is, for continuous functions, checking to see if resulting values are within interpolated values of nearby test points.
- e) Data diversity, that is to see if the output differs significantly from the input.
- f) Dual programming, which is to develop two versions of the program, each by an independent team, with the hope that they will not make identical errors.

It is obvious that there is no single method that is perfectly automatic and able to identify any wrong output. Furthermore, some methods are only applicable to a certain type of software. A properly selected mix of some of the above approaches will reduce the number of manual checks that would otherwise be required.

In our work, besides manual checks, the last approach, dual programming, was applied in our demonstration program. Also in most of the programs developed these days, embedded assertion is adopted too. But because this is part of the work that should be done during programming and have become every programmer's custom, we did not cover it in our work. The details about how we took advantage of dual programming are discussed in Chapter4.

3.3.4.5. Stopping Rule Base on Completeness

The traditional stopping rule states that if the estimated reliability reaches the required reliability value the testing process can be terminated. The number of failure-free test cases is connected with the reliability objective with some reliability estimation model. In this project, we would like to approach the stopping rule decision from a different perspective. Imagine, what we could do, if we were not to have a required reliability value corresponding to our reliability estimation model. Or, if we want to put the reliability requirement on hold for the moment and consider how reliable we want the safety-critical software to be. Ideally, we want it to be failure free. In other words, we want to test the software as complete as possible.

Because we intend to test the software completely on an executable flowpath basis, we need to decide upon the final level of flowpath coverage completeness before at which we can stop testing. As we found in our demonstration work, being exhaustive in

terms of flowpath coverage is prohibitively expensive even for relatively simple system control software. Use of some reasonable assumptions and simplifications becomes essential in making the testing strategy pragmatic. In the following two subsections, a phenomenon favoring the possibility of stopping the testing process early is discussed first; followed by an important assumption used to bridge the gap between the testing termination point and the executable flowpath complete coverage point.

3.3.4.5.1. Input Saturation Effect

Although we intend to attain complete testing in the first place because of the safety-criticality nature of the software that we are working on, it is also true that the running cost of such testing is huge. Another option would be that we may have to sacrifice some of the completeness in order to achieve practicality. But how can we trust the incompletely checked software? We have to find a way to show that in spite of the incompleteness, it is safe enough to use. Recent research work shows that software developed by using formal methods exhibit an early saturation effect during testing. Thanks to this effect, random input sampling could be used to achieve nearly complete flowpath coverage, even for some flowpaths that are not reachable by formal exhaustive input data identification. In this manner, we avoid the theoretical problem of incompleteness, provided that testing continues until after the saturation point. Such random sampling is rapid, consumes little memory, is simple to implement, and can handle very large formal models.

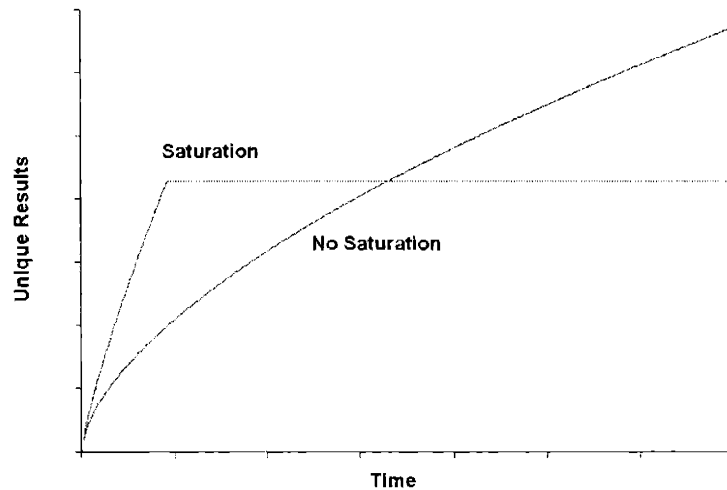


Figure 3-6 Illustration of Saturation Effect in Software Testing

Saturation effect in our research context is visualized in Figure 3-6. In the picture, most of the flowpaths have been reached very early in the random input sampling and testing process. The assessment methods that exhibit a saturation effect support an early stopping rule. That is to say, the quicker the saturation point is reached the earlier the test can be stopped, because by then we should have covered most of the executable flowpaths. It is claimed that assessment methods that use early stopping rules run the risk of false positives, i.e., reporting that no faults are present when further assessment would have found them. Hence, the early stopping rule can only be endorsed for sampling that exhibits adequacy and flat plateau properties. Here, we briefly describe these two properties, followed by discussions about whether our safety-critical software testing process satisfies these requirements. And if not, what we should do to solve the problem.

Adequacy: An adequate assessment method does not fail to recognize faults in states visited (covered) prior to the occurrence of saturation. This is to say that whenever we have tested a flowpath we are always able to detect the error if there is any. For the majority of software errors, it is true. But still there is a sizeable possibility that some errors might not be discovered the first time that a flowpath is tested. Because of this reason, as described in much more detail in Section 3.4.4.2.3.1, we divide the errors into two types. If a flowpath contains a type I error, it is always discovered during the first time that the flowpath is tested. But if there is a type II error, the possibility of detecting

the error is the same during any single visit to the flowpath. The event of finding a type II error during any single visit to the flowpath is independent of any other visit to the same flowpath. In this work, we use Bayesian updating to estimate the percentages of type I errors, p_I , and that of type II errors, p_{II} , as well as the possibility of finding a type II error during a single visit to an error-prone flowpath, $p_{f,II}$. It is clear that only the error-triggering flowpath information could be used for this estimation purpose. The safety-critical software developed via use of formal methods can provide us with very limited data points. We argue that errors within all similar programs in terms of their testability³ should also have these two types of errors with similar corresponding probabilities: P_I , P_{II} , and P_f . This implies that information from all such testing processes could be used to update distributions of these probabilities. The updated result at any stage could always serve as prior information for further updating.

Flat plateau: If the saturation curve results in a flat plateau, then additional errors, if present in the unseen portion of the formal model, are not detected. However, because of the flatness of the plateau, they are most likely to remain unnoticed upon system deployment, unless the associated operating regime undergoes drastic changes. Even though studies [Menz00,Menz02] show that many formal models feature such flat plateaus, there still exist cases where the flat plateau is not very clear. If this happens, we should draw the real flowpaths vs. number of input samplings curve, from which we can estimate how far away we are from the saturation point. A proper confidence level should be associated with that estimation. Again the detailed estimation process is delayed until the reliability estimation session.

3.3.4.5.2. Continuity Assumption Incorporation

From the reliability estimation point of view, in order for us to stop testing before complete flowpath coverage is achieved, we have to use some measurement of the complexity of the whole program. The information obtained from the tested flowpaths is

³ According to the IEEE Glossary of Software Engineering Terminology, testability is defined as “the degree to which a system of components facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.”

the sole base of this measurement. Continuity assumptions from the tested flowpaths of the software to the untested flowpaths are needed to estimate reliability of the entire software. In order to achieve this, the tested flowpath indexes vs. the number of unique input sampling data that has triggered each of these flowpaths curve are plotted. The probability that an untested flowpath will be visited is estimated by extrapolating this curve. To extend the reliability information of the tested portion over to the untested portion of the software for reliability prediction, we assume that the reliability of every unchecked flowpath has the same value as the average reliability of the checked flowpath given that no modification has been made and both portions were designed and developed by the same group of personnel under the same programming environment. In this section, we only intend to cover these assumptions briefly in order to make the claimed testing termination before the complete flowpath coverage justifiable in every respect.

3.3.4.6. Testing Result Recording

Dual programming is the main output checking method. Our approach is a little different from conventional dual programming, i.e. we take a developed and thoroughly tested program based upon the same algorithm as the software under consideration as the oracle program. Most of the time⁴, if we notice any difference between the outputs from the two programs, we believe that the output from the oracle program is correct. We then claim the discovery of an error in the software under testing at this point. For this reason, we always record results from the two programs in the same format, which provides considerable convenience when making output comparisons.

⁴ There is a very small possibility that the oracle program gives the wrong output.

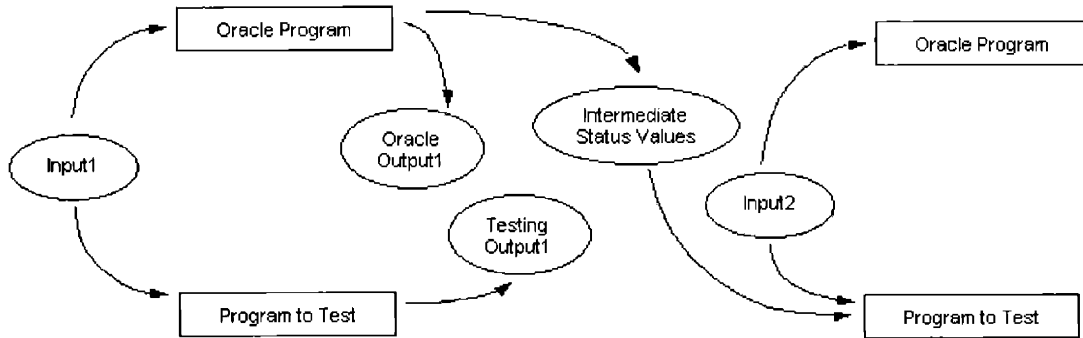


Figure 3-7 Conversing of Intermediate Status Values into Input Values

An important characteristic of signal processing programs is that adjacent runs of the program are sequentially dependent via some intermediate status values. Because we have to remember the output of every run of the program, we cannot let it run continuously as is the case during field use. This requires that we turn the intermediate status variables, unseen to an outsider during operational time, into input variables. The values of such variables are those of the intermediate status variables from last run of the program. This additional process during testing forces the testing process to follow a special sequence. A series of randomly sampled input data are first put into the oracle program. Outputs and intermediate status values are recorded after each run of the oracle program. Then, both the same set of inputs and the intermediate status values are put into the program under testing, outputs are recorded after each run of that program.

There is another piece of information that needs to be remembered after each test is the flowpath that was followed in this test. This is not necessary during normal test, when one only intends to decide if the given input could lead to any failure of the software. However, in our approach, we need this information due to two reasons. First, we need it to decide how many flowpaths have been tested. According to the coverage percentage and how far away the test is from the saturation point, we can decide if we can terminate our testing process in terms of completeness. Second, we need the flowpath information to estimate the reliability of every particular flowpath and thereby further

estimate the reliability of the whole program. The method used to calculate the estimation is covered in detail in the reliability estimation section.

3.3.4.7. Reassessment After Modification

Several of the reliability growth models assume that the modification of a faulty point in the software does not introduce new errors. Under this assumption, reassessment is not necessary as long as the program is not modified for reasons other than the discovery of a fault, such as the incorporation of new features or a better algorithm. However, this assumption is not sound enough to be applied to the safety-critical software. Even though the possibility of introducing a new error is very small⁵, we cannot take that risk because of the possible severe consequences. Reassessment in this project is always required.

Partial retest of those affected portions of the software might be suggested as a means to avoid the cost of assessment starting from scratch in the event of a debugging action following a failure. The decision about which one should be adopted depends only upon the expense involved because both of them can guarantee that none of the previously tested input will cause a failure. The extra work brought about by the partial retesting approach is the identification of the affected portion of the program and which previously used input data are related to that portion. If the identification work is more complicated and time consuming, the complete will be the better choice. Furthermore, if the modification is made at a very high level of the whole software structure, most of the flowpaths are affected and most of the input data should be used for retesting.

In this project, because of the relative simplicity of the software under consideration and the high confidence requirement, we decided to use the full reassessment approach. Because the testing process is automated and the identification of the affected input data is therefore comparatively harder, partial retesting becomes more expensive. It is more costly to differentiate the affected paths from the unaffected ones than to test all the test data used before, let alone the probability of having some

⁵ In this project, we did not encounter such cases. In others word, every time we detected an error and made corresponding modifications, the program always behaves correctly with all the previous used input data.

error in the new tool. In the future, if more complicated software is to be tested, further efforts should be made so that the software is more modular. Hence, the related structure and input data would be much easier to identify.

3.3.4.8. Nodal Coverage Checking

As discussed in the white box testing techniques that branch coverage is a far less easy job than path coverage approach. Also, if complete path coverage testing has been achieved on a program, branch coverage testing must have also been achieved. This is to say branch or nodal coverage testing is a pre-requisite of path coverage testing. One of the results of nodal coverage marking process is identification of any untested node. Also, if a node is a decision node and has not been visited, all its child nodes must have not been visited either. After this process is done and if the number of nodes unvisited is not very large, manual checking can be done on those specific nodes. Those nodes tend to be ones that deal with abnormal scenarios, e.g., a careless input from the operator that is out of the specified range. If this is true, the corresponding input data should be designed especially to visit these nodes. To achieve this, both the programmer and the designer, who writes the specifications, should be present. This is the same job that the tester is supposed to do in an idealized white box testing approach, that is, part of the program structure is identified or located first and input data, which can lead an execution toward that structure, are selected. The situation also exists that a node, which can never be visited, is found in the unvisited node group. If this happens, that node might better be removed from the program because this is a useless node.

3.4. Reliability Estimation Methodologies

3.4.1. Software Reliability Model Review

Software reliability is one of the most important parameters of software quality. In the 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 often than not simply set their tolerance to be error-free, especially under safety-critical circumstances. 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, which is a design defect that is activated by a certain input to the code during its execution. During our study, we consider a more simplified situation than that mentioned above in the two reliability definitions. Instead of considering a continuous time software system, we view our target program as a discrete demand-based system. Instead of 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 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. The objective is to determine 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 either from testing or from operation to measure software reliability. Software reliability models are classified as being either black box or 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 is given. Both from their names and their essential characteristic differences, they seem to have exactly the same relationship as black box testing and white box testing methods. But there is a significant difference in that these two types of reliability analysis methods are targeting two different levels of the software systems (The black box method can be applied to any software from its bottom level while 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. The white box reliability models uniformly assume the component reliabilities are available and are always working on top of the black box models.

3.4.1.1. Black Box Reliability Models

In the software development process, it is very typical to end up containing a product with some defects, i.e. faults, or bugs. For a specific input to the software, these faults are activated and result in a deviation of the software behavior from its specified behavior, i.e. a failure. It is common to demand that all known faults be removed, especially in the case of safety-critical software. This means that if there is a failure found during testing, the offending fault must be identified and removed without introducing new bugs. Thus it is a fundamental for all black box models to assume that every fault is perfectly fixed, i.e. the process of fixing a fault does not bring about any new fault, and software reliability increases. If the failure data is recorded either in terms of number of failures observed per test or given time period or in terms of the number of tests or time between failures, statistical models can be used to identify the trend in the recorded data, reflecting the growth in reliability. Such models are known as Software Reliability Growth Models (SRGMs) or growth models in general. They are used to both predict and estimate software reliability. All SRGMs are of the black box type because they only consider failure data, or metrics that are gathered if test data is not available. Black box models do not consider the internal structure of the software in reliability estimation and are called such because they consider software as a monolithic entity, a closed box. In the subsequent portion of this section, several basic black box models are reviewed. There are many other existing black box models, but all are extrapolations of these fundamental models.

3.4.1.1.1. Terminologies in Black Box Reliability Model

Some of the terms commonly used in the context of Software Reliability Growth Models (SRGM) are listed in Table 3-1. Other terms are also used in different models and are explained as they are encountered.

Term	Explanation
$M(t)$	The total number of failures experienced by time t .
$\mu(t)$	Mean value function for an SRGM. This represents the expectation of the number of failures expected by time t as estimated by the model. Therefore, we have $\mu(t) = E[M(t)]$
$\lambda(t)$	Failure intensity function, representing the average number of failures per unit time predicted by the model. It is the derivative of the mean value function, i. e. $\lambda(t) = d\mu(t) / dt$
$Z(\Delta t t_{i-1})$	Hazard rate of the software, which represents the probability density of experiencing the i^{th} failure between t_{i-1} and $t_{i-1} + \Delta t$ given that $(i-1)^{\text{st}}$ failure occurs at t_{i-1} .
$z(t)$	Per-fault hazard rate, which represents the probability that a fault, that had not been activated so far, will cause a failure instantaneously when activated. This term is usually assumed to be a constant (ϕ) by many of the models.
N	Initial number of faults present in the software prior to testing.

Table 3-1 Terminology Common to Black Box Reliability Models

Data that are generally supplied to SRGMs are either the times at which failure occurs $\{t_0 = 0, t_1, t_2, \dots, t_n\}$ or elapsed time between failures $\{x_1, x_2, \dots, x_n\}$, $x_i = t_i - t_{i-1}$. There is an assumption common to all the models presented in this section, i.e., the failures are independent of each other. Though, there are frameworks [Triv01] that incorporate statistical dependence between failures, they are not covered in this review.

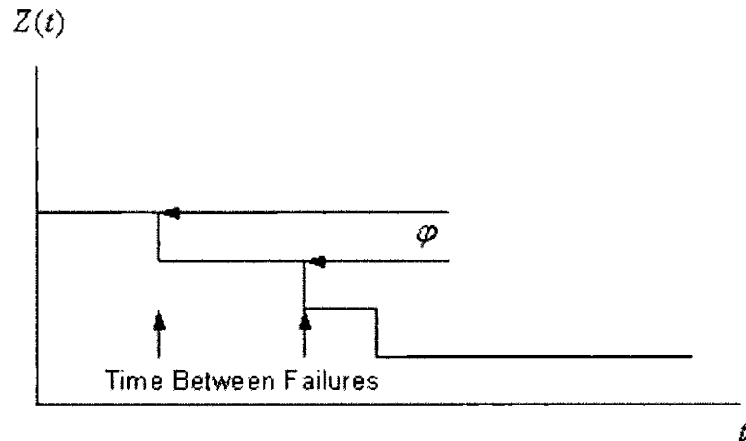
Four classes of black box reliability models are briefly reviewed here: failure rate model, error seeding model, curve fitting model, and Bayesian model. In the following subsections, each class is described briefly followed by an example model within that class.

3.4.1.1.2. Failure Rate Model Class

The failure rate models emphasize the per fault failure rate during the failure intervals. The models in this group try to capture the how the failure rate changes as more faults are detected and corrected. Estimations on the total number of errors in the software and failure rate are made based upon the data collected during the 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.

One of the earliest software reliability models, the Jelinski-Moranda (J-M) model (Jelinski, 1972) is representative of the failure rate model class. Many existing software reliability models are variants or extensions of the J-M model.

The elapsed time between failures is taken to follow an exponential distribution with a parameter that is proportional to the number of remaining faults in the software. With a constant per fault hazard rate ϕ after detection of the $(i-1)^{th}$ failure⁶, the mean time between failures at time t ($t_{i-1} < t < t_i$) is $1/\phi(N-(i-1))$. The figure below illustrates the impact that finding a fault has on the overall hazard rate. As each fault is discovered, the hazard rate is reduced by the proportionality constant ϕ , the per fault hazard rate. The impact of each fault removal on the hazard rate $Z(t)$ is the same.



⁶ We assume a one to one relationship between faults and failures while presenting the existing software reliability models. Thus, failure and fault will be used interchangeably throughout this section.

Figure 3-8 Hazard Rate of J-M Model

The assumptions in this model are:

- The data to estimate software reliability are collected while the software is operated in a manner similar to that for which the reliability estimation is to be made.
- Each fault in the program has the same chance of being encountered.
- The failures, when the faults are detected, are independent.
- The program contains N initial faults, where N is unknown but finite and fixed constant.
- Time intervals between occurrences of failures are independently and exponentially distributed random variables.
- The fault that causes a failure is instantaneously removed and no new faults are introduced when the detected faults are removed.
- The software failure rate, during a failure interval, is constant (φ) and is proportional to the number of faults remaining in the program.

The above assumptions lead to the hazard rate $Z(\Delta t | t_{i-1})$, after removal of the $(i-1)^{th}$ fault, being proportional to the number of faults remaining in the software $N - M(t_{i-1})$ and the per fault hazard rate φ . That is:

$$Z(\Delta t | t_{i-1}) = \varphi[N - M(t_{i-1})] = \varphi(N - i + 1) \quad \text{Eq. 3-1}$$

If the time-between-failure occurrences are $X_i = T_i - T_{i-1}, i = 1, \dots, n$, then the X_i 's are independent exponentially distributed random variables with mean $\frac{1}{\varphi(N - (i-1))}$. That is:

$$f(X_i | T_{i-1}) = \phi[N - (i - 1)] \exp(-\phi[N - (i - 1)]X_i) \quad \text{Eq. 3-2}$$

The mean value function and the failure intensity function for this model are

$$\mu(t) = E[M(t)] = N(1 - e^{-\phi t}) \quad \text{Eq. 3-3}$$

$$\lambda(t) = N\phi e^{-\phi t} = \phi(N - \mu(t)) \quad \text{Eq. 3-4}$$

This is clearly a finite failures model as $\lim_{t \rightarrow \infty} \mu(t) = N$. Software reliability obtained from this model can then be expressed as

$$R(t_i) = e^{-\phi(N-i+1)t_i} \quad \text{Eq. 3-5}$$

The data requirements to implement this model are: the elapsed time between failures x_1, x_2, \dots, x_n or the actual times that the software failed t_1, t_2, \dots, t_n , where $x_i = t_i - t_{i-1}$. There are two unknown parameters in the model, the initial number of faults present in the software, N , and the per fault failure rate ϕ . Using the Maximum Likelihood Estimation (MLE) method, we can estimate these parameters from the joint density of the X_i 's as:

$$\hat{\phi} = \frac{n}{\hat{N} \left(\sum_{i=1}^n x_i \right) - \sum_{i=1}^n (i-1)x_i} \quad \text{and} \quad \text{Eq. 3-6}$$

$$\sum_{i=1}^n \frac{1}{\hat{N} - (i-1)} = \frac{n}{\hat{N} - \left(\frac{1}{\sum_{i=1}^n x_i} \right) \left(\sum_{i=1}^n (i-1)x_i \right)} \quad \text{Eq. 3-7}$$

where,

n is Total number of failures encountered during data collecting process

x_i is	The time interval between i^{th} failure and $(i-1)^{\text{st}}$.
\hat{N} is	MLE estimated total number of faults in the software
$\hat{\phi}$ is	MLE estimated constant per fault hazard rate

First value of N is estimated by the second equation. Then, the solution is put into the first equation to find the MLE of ϕ . The reliability measure in Eq. 3-5 can be derived by replacing the quantities N and ϕ with the corresponding MLE of \hat{N} and $\hat{\phi}$.

Though the J-M model has now largely been replaced and the importance of the J-M model is mainly in setting the framework for future work in this modeling area.

Other reliability models in this failure rate category are:

- Schick and Wolverton model (Schick, 1978)
- Jelinski-Moranda geometric model (Moranda, 1979)
- Moranda geometric Poisson model (Littlewood, 1979)
- Negative-binomial Poisson model
- Modified Schick and Wolverton model (Sukert, 1977)
- Goel and Okumoto imperfect debugging model (Goel, 1979)

3.4.1.1.3. Error Seeding Models

This class of reliability estimating models estimates the total number of errors in the given software by intentionally introducing bugs. Tests are performed on the software, which contains both inherent errors and induced errors. As testing goes 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 assumptions made in this model are:

- The data to estimate software reliability are collected while the software is operated in a similar manner as that for which the reliability estimation is to be made.

- Each fault (either inherent or seeded) in the program has the same chance of being encountered.
- The failures, when the faults are detected, are independent.
- The program contains N initial faults, where N is an unknown but finite and fixed constant.
- The fault that causes a failure is instantaneously removed and no new faults are introduced when the detected faults are removed.

Mill's error seeding technique (Mills, 1970) is a method to estimate the total number of errors in software under consideration by inducing seeded errors into the program. Assume there are N inherent errors in total and n_1 seeded errors in the software. If the probabilities of detecting both types of errors are the same, the probability that there are k induced errors among the r errors corrected can be calculated according to hypergeometric distribution as:

$$P(k, r; n_1, N) = \frac{\binom{n_1}{k} \binom{N}{r-k}}{\binom{N+n_1}{r}}$$

Eq. 3-8

where,

N is the total number of inherent errors in the program,

n_1 is the total number of seeded errors in the program,

r is the total number of seeded errors corrected,

k is the total number of errors corrected,

$r - k$ is the total number of inherent errors corrected.

In equation (3-7), the only unknown value is N . It can be estimated according to MLE as:

$$\hat{N} = [N_0] + 1 \quad \text{Eq. 3-9}$$

$$N_0 = \frac{n_1(r-k)}{k} - 1 \quad \text{Eq. 3-10}$$

If N_0 is an integer, both N_0 and $N_0 + 1$ are MLE estimates of N .

Unlike the failure rate model, this model does not estimate reliability in a time-dependent manner. The only information obtained is the total number of errors in the software. The important assumption made in this model is that there is equal opportunity to encounter either an inherent or an induced error. This makes the error seeding process very difficult. If no data have been collected about any of the inherent errors, how can we create some errors that are similar to the inherent errors?

The other models within this category include:

- Cai's model (Can, 1998)
- Hypergeometric distribution model (Tohma et al., 1991)

3.4.1.1.4. Curve Fitting Models

Curve fitting models use statistical regression methods 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 of model can be divided into several groups based on the software reliability parameter that is sought, e.g. total number of errors in the software, the complexity measure of the software or the failure rate of the software, etc.

The data required within this model group are very different depending on the proposed relationship function type. Because this group of models is based more on the experimental observations than on the underlying physics, emphasis is on the patterns noted from the data. Therefore, certain techniques are only applicable to a specific type of software.

3.4.1.1.5. Bayesian Models

Both of the two reliability models used in this project are fundamentally Bayesian models. This group of models views reliability growth and prediction in a Bayesian framework rather than in the traditional ones considered in the previous sections. The power of the Bayesian updating strategy is that it can include various forms of information, both subjective expert judgments and objective experimental results, in one model.

The previous models allow changes in the reliability estimation 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.

Most of the traditional models also look at the impact of each fault as being of the same importance. The Bayesian model reflects the belief that different faults have different impacts on the reliability of the program. If we have a program that has several faults in seldom used code, the program is not necessarily less reliable than the one that has only one fault in the part of code that is used very often. The number of faults becomes less important in the Bayesian model compared to the actual impact made by the faults to be implemented.

The prior distribution, reflecting the view of the model parameters from past data, is an essential part of this methodology. One can incorporate past information, projects of similar nature, for example, in estimating reliability statistics for the present and future. This distribution is simultaneously one of the Bayesian's framework strengths and weakness. Specifying a meaningful full prior distribution over all variables sometimes would make the model too difficult.

Most other models view the value of the hazard rate to be a function of the number of faults remaining. In contrast, the L-V model takes it as a random variable. All the information on hand could be used to update the distribution of that random variable.

As a result, the reliability or hazard rate estimation can be expressed in a statistical manner.

The basic idea on the mathematics behind this theory is as follows. Suppose we have a distribution for our reliability data that depends upon some unknown parameters, $\vec{\xi}$, that is, $f_T(\vec{t} | \vec{\xi})$ and a prior $g(\vec{\xi}, \vec{\varphi})$ that reflects our view on those parameters $\vec{\xi}$, from historical data. Once additional data have been gathered through the vector \vec{t} , our view of the parameter $\vec{\xi}$ changes. That change is reflected in the posterior distribution, which is calculated as

$$h(\vec{\xi} | \vec{t}, \vec{\varphi}) = \frac{f_T(\vec{t} | \vec{\xi})g(\vec{\xi}, \vec{\varphi})}{\int \dots \int f_T(\vec{t} | \vec{\xi})g(\vec{\xi}, \vec{\varphi})d\vec{\xi}} = \frac{f_T(\vec{t} | \vec{\xi})g(\vec{\xi}, \vec{\varphi})}{f_T(\vec{t}, \vec{\varphi})} \quad \text{Eq. 3-11}$$

Various estimates of $\vec{\xi}$ can be obtained using the posterior distribution, thereby leading to reliability estimates involving $\vec{\xi}$. As the testing continues, the distribution of $\vec{\xi}$ could be updated again and again by using the previous posterior distribution as current prior distribution.

Another strong point of this model is that it can account for fault generation in the fault correction process by allowing reliability to decrease. Because of uncertainty, after each fault removal, the new version could be better or worse than the predecessor. Thus, another source of variation is introduced.

3.4.1.2. White Box Reliability Models

The white box software reliability models consider the internal structure of the software in reliability estimation as opposed to the black box models which only model the interactions of software for 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 because the 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 the development of techniques to analyze the performance of software built from reused and Commercial-Off-The-Shelf (COTS) components, to perform sensitivity analyses, (i.e. to study the variation of application reliability with variation in component and interface reliability, and to identify of critical components and interfaces.

The first step in white box reliability models or architecture-based software models is to decompose the software. A component of the software is conceived as a logically independent unit of the system that performs a well-defined function. The level of decomposition depends on the tradeoff between the number of components, their complexity, and the available information about each component. Besides components, the interfaces among components should be identified through the software architecture. Interactions occur only by transfer of execution control. When the possible interactions are specified, non-zero transition probabilities should be assigned to each interaction by analyzing the program structure and using operational profiles. The next step, failure behavior, is defined and associated with the software architecture. Failure can happen during execution of any component or during a control transfer between two components. The failure behavior of the components and of the interfaces can be specified in terms of their reliabilities or failure rates. Software reliability growth models are applied to estimate reliability for each software component by exploiting components' failure data obtained during testing. Interface failures happen separately from component failures, and can be observed through integrated testing.

The final step in this reliability group is to combine the software structure together with its components' failure behaviors. According to the methods to make the combination, there are two types of white box reliability models: the state-based model group and the path-based model group. They are presented separately in the following subsections.

3.4.1.2.1. State-Based Model

In this model class, a control flowgraph is used to represent the software architecture. Software reliability is estimated analytically. These models assume a control flowgraph has a single entry and a single exit node that represents components at which execution begins and terminates, respectively.

The Cheung model is one of the earliest models that relate software reliability to component reliabilities. The transfer of control among components is described by a transition probability matrix $P = [p_{ij}]$, where p_{ij} is the probability that the program transits from component i to component j . Each component is assumed to fail independently with probability $1 - R_i$, where R_i is the probability that component i performs its correct function. In order to relate the overall reliability with the component reliabilities, two absorbing states C and F are added, representing the correct output and failure. The transition probability matrix is modified to \hat{P} as follows. Each element p_{ij} is replaced by $R_i p_{ij}$, which represents the probability that the component i produced the correct result and the control is transferred to component j . From the exit n , a direct edge to state C is created with probability R_n to represent to correct execution. The failure of component i is considered by creating a direct edge from i to F with transition probability $(1 - R_i)$. The reliability of the program is the probability of reaching the absorbing state C of the Discrete Time Markov Chains.

In this type of model, it is assumed that component failure will eventually lead to system failure and every of component fails independently.

3.4.1.2.2. Path-Based Model

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. In contrast, components or nodes are examined in state-based models.

Though named as “Component Based Reliability Estimation”, Krishnamurthy And Marthur’s model is a typical path-based model. They experimentally investigate a method for estimating the reliability of a system given the reliabilities of its components and of their interfaces with other components. Specifically, sequences of components along different paths are observed during the testing. The component trace of a program P for a given test case tc , denoted by $M(P,tc)$, is the sequence of components m executed when P is executed against tc . A sequence of components along a path traversed for test case tc is considered as a series system, and, assuming that components fail independently of each other it, follows that the path reliability is:

$$R_{tc} = \prod_{\forall m \in M(P,tc)} R_m \quad \text{Eq. 3-12}$$

This is to say that the reliability of the path in P traversed by test case tc is the product of the component reliabilities. The reliability estimation for program P with respect to a test set T is given by the average of all the path reliabilities as:

$$R = \frac{\sum_{\forall tc \in T} R_{tc}}{|T|} \quad \text{Eq. 3-13}$$

3.4.1.3. Reliability Model Conclusion

From the literature, the black box models are the ones traditionally used to quantify reliability for less complex software system. Compared to the white box models, they are simpler because no structural information of the software is needed. As a result, the black box approaches provides very little information indicating the completeness of the test. Almost no significant software features can be revealed by this group of reliability estimation models.

In the white box models, the architecture of the software is identified, not in the sense of traditional software engineering architecture but rather in the sense of interactions between components. The interactions are defined as control transfers, essentially implying that the architecture is a control-flow graph where the nodes of the graph represent modules and its transactions represent transfer of control between the modules. The failure behavior of these modules (and the associated interfaces) is specified in terms of failure rates or reliabilities (which are assumed to be known or are computed separately from SRGMs). The failure behavior is then combined with the architecture to estimate overall software reliability as a function of component reliabilities. Compared to the black box models, they are better approaches to answer the completeness question and reveal important software feature. But because this group of methods is designed for more complicated, modulated software systems, its basic assumption is that the component reliabilities are known. Such models simply ignore the issue of how these values could be determined, which is still an open research issue.

3.4.2. Two Reliability Estimation Methods Overview

In this project, we tend to design reliability estimation methods that combine strengths of both black and white box approaches. We want keep the simplicity of the black box methods and the software feature indication capability of the white box methods.

We have developed and demonstrated two reliability estimation methods: nodal coverage based reliability estimation model and flowpath coverage based reliability estimation model. Both models are structure based models. No manual architecture identification process, as in the white box models, is needed. The nodes and flowpaths that have already existed in the program when testing starts serve as software components naturally. The component reliabilities are estimated via Bayesian updating method. It is the most proper method for our purpose because it works perfectly even when no failure case has been revealed and such no failure tests will dominate the entire testing process for the safety-critical software systems.

The nodal coverage based reliability estimation method is a simpler version of the flowpath coverage based method. It can be used as a quick and approximate substitute of the flowpath coverage based method.

3.4.3. Complete Nodal Coverage Reliability Estimation

3.4.3.1. Method General

Because of the safety-criticality of the software on which we are working, the goal of reliability estimation is to provide a more precise software reliability measurement technique. It should be able to take advantage of all the available information can provide. Also, it should be able to produce constructive feedback for 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 that 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, complete nodal coverage testing is achieved. Testing results for each node and also its visiting frequency during testing are 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 a visiting frequency weighed average of all nodes multiplied by the average number of nodes visited per execution.

In the subsequent portions of this section, the initial motivations, the assumptions and method itself are presented in more details.

3.4.3.2. The Initial Incentive for Nodal Coverage Based Approach

The nodal coverage-based reliability estimation method is a by-product on our evolution toward a path-coverage based reliability estimation method. It is relatively less demanding to achieve so that it can be applied to more complicated software than the programs that we studied in the work reported here. We notice from our demonstration work that, although the nodal approach is not an accurate formulation and does not use as much information as the path approach, it still gives us a reasonable approximation of the software's reliability.

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, many problems concerning feasibility and automation appeared very quickly. The method of our work presented in this report is much different compared to the original design of our work for this reason. It is the influence of the demonstration work that directs us toward the final estimation approaches that are described in this report. This complete nodal coverage reliability estimation technique is one of those ideas that were brought about in this way.

In some sense, the nodal coverage reliability estimation is a simplified version of path coverage reliability estimation approach. At the starting point of this project, because of the safety-critical feature of the target software used in nuclear power plants, we chose the goal of testing the software as completely as possible and to maintain feasibility at the same time. These goals lead us to work toward a complete path coverage testing approach, which implies the reliability estimation technique is based on the paths also. When we were working on programs that were relatively simple and testable, we believed that we could cover most of the paths in 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 that we have not met before during the testing process. We do not identify any paths before testing. The way we recognize, or identify a path is through testing. Every time a test is done, its flowpath is recorded. If we cannot find the same path in the group of paths that we have tested before, it is announced as a new path. At this stage, we had to answer a fundamental

question. Namely how many paths in total do we have in the target program? We have to say that we do not know. There are lots of existing paths that may not be reached by real input data.

We decide to make our task simpler at first by trying to test all of the nodes at least once. 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. Because we assume that 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 when calculating the overall software unreliability. The reliability of the software is unity minus the 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.

3.4.3.3. Methodology Details of the Nodal Coverage Based Reliability Estimation

3.4.3.3.1. 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.

Assumption 1):

Testing is representative of actual use. The assumption says that 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. Because in this nodal coverage reliability estimation approach, we assume that any error in a node is not path dependent. That is, if there is an error in a node, all the paths that pass through that node will fail. Thus the first importance factor of a node is very simple. The more often it is visited, the more important the node is. Now, if we try to perform a test as close to the true usage situation as possible, the first importance factor for every node can be approximated by the visiting frequency of each node during the 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.

Assumption 2):

Faults are of the same severity. Another importance factor attached to each node concerns how bad 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 requirements, the software specifications and the code simultaneously and relate each of the actual functions to a node, or to 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 in this study, we are not attacking this problem because of its hard-to-be automated nature. Also, because our software under question is relatively small and simply structured, there are not many functions involved. This leads to a very small number of different importance levels in terms of nodal failure consequences. For the above reasons, we assume that every node has the same failure consequence severity.

Assumption 3):

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 the nuclear industry has very high safety criterion that demand all detected faults be removed. We ensure the realization of this requirement by full retesting. If an error is encountered, immediate correction should be made and all the previously used input data should be used to test the modified software.

Assumption 4):

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. 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 the 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 one assumption is very nodal related and is specific to this reliability estimation technique. Both of them have very solid foundations 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 based only on the facts that 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 is a result of the Bayesian updating method. Only facts about how many times that a node has been tested with or without failures 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 of introducing new errors while correcting the detected ones. That is, new faults can be introduced as a result of fixing existing faults. As opposed to many reliability estimation methods, we do take into consideration the possibility of introducing new errors because of 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 discovery of the latest error. But it is not totally sure to say there are no new errors brought about by the modification to the untested part of the software. We still believe that software reliability grows because of 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 probabilistic manner. If an error is detected during the k^{th} visit to that node and modified immediately, we put the new information that we have tested the node k times without an error, rather than $(k-1)$ times without an error, into the Bayesian updating model used 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 because of 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 that 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 than the assumptions.

3.4.3.3.2. The Formulas

This nodal coverage has a pre-requirement that all nodes be tested. This means complete branch coverage should have been achieved and every direction of each decision option within the program be tested at least once. Upon completing that testing

process, unreliability is estimated for each of the nodes in the program based on Bayes theorem. As testing goes on, unreliability is updated using the Bayesian updating method for every node. The overall unreliability of the software is approximated by the weighted summation of that of all of the nodes within the software multiplied by the average number of nodes visited per test. In other words, the software reliability is the product of an average unreliability considering all nodes and an average number of nodes reached per execution. The weight for each node, used while calculating the average unreliability, is proportional to the visiting frequency of that node during testing. If we use the arithmetic average formula, the unreliability of the software is:

$$\theta_n = \frac{\sum_{i_n=1}^{N_n} p_{visit,i_n} \bar{\theta}_{i_n}}{\sum_{i_n=1}^{N_n} p_{visit,i_n}} \times \bar{x}N_n = \frac{\sum_{i_n=1}^{N_n} f_{visit,i_n} \bar{\theta}_{i_n}}{\sum_{i_n=1}^{N_n} f_{visit,i_n}} \times \bar{x}N_n \quad \text{Eq. 3-14}$$

where,

- θ_n is the software unreliability estimated by complete nodal coverage reliability estimation approach,
- N_n is the total number of nodes in the software,
- p_{visit,i_n} is the probability of node i_n being visited during real operation,
- f_{visit,i_n} is the visiting frequency of node i_n during testing process,
- $\bar{\theta}_{i_n}$ is the mean unreliability of node i_n , estimated by Bayesian updating method,
- x is the average fraction of nodes visited per test.

If instead, the geometric average formula is used, the software unreliability can be estimated as:

$$\theta_n = \exp\left[f_{visit,i_n} \ln(\bar{\theta}_{i_n})\right] \times \bar{x}N_n \quad \text{Eq. 3-15}$$

The average nodal unreliability is the geometric average of all the node unreliability, i.e.:

$$\bar{g}_n^{\sum f_{visit,i_n}} = \prod \bar{\theta}_{i_n}^{f_{i_n}} \quad \text{Eq. 3-16}$$

Due to the relation between θ_n and $\bar{\theta}_{i_n}$, we choose the arithmetic average formula shown in Eq. 3-14. See the Appendix for the details.

Among the above parameters,

$$P_{visit,i_n} = f_{visit,i_n} = \frac{n_{ic,i_n}}{N_{ic}} \quad \text{Eq. 3-17}$$

where

n_{ic,i_n} is the number of test cases that bypass node i_n ,
 N_{ic} is the 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 the 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:

$$P(R_{i_n} = r) = C_n^r \theta_{i_n}^r (1 - \theta_{i_n})^{n-r} \quad \text{Eq. 3-18}$$

Within the Bayesian framework one represents one's prior knowledge about the parameter of interest, in this case θ_{i_n} , by the prior distribution. There are advantages to 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 changes as one receives additional information. Here, the conjugate distribution chosen is the $Beta(a,b)$ distribution:

$$f(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b-1}}{B(a,b)} \quad \text{Eq. 3-19}$$

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, the "ignorance" uniform prior with $a = b = 1$ can be used:

$$f(\theta_{i_n}) = 1 \quad \text{Eq. 3-20}$$

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 $Beta(a + r, b + n - r)$:

$$f(\theta_{i_n}) = \frac{\theta_{i_n}^{a+r-1} (1 - \theta_{i_n})^{b+n-r-1}}{B(a+r, b+n-r)} \quad \text{Eq. 3-21}$$

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

$$f(\theta_{i_n}) = \frac{\theta_{i_n}^r (1 - \theta_{i_n})^{n-r}}{B(1+r, 1+n-r)} \quad \text{Eq. 3-22}$$

There is a non-obvious assumption behind the Bayesian updating method. The underlying parameter or parameter distribution is fixed while being updated by use of the 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 specific, 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 the unreliability of node i_n as:

$$f_0(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b-1}}{B(a, b)}, \quad a > 0, \quad b > 0 \quad \text{Eq. 3-23}$$

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

$$f_0(\theta_{i_n}) = 1 \quad \text{Eq. 3-24}$$

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

$$f_1(\theta_{i_n}) = C_1(1-\theta_{i_n})^{\Delta s_1} f_0(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1}(1-\theta_{i_n})^{b+\Delta s_1-1}}{B(a, b + \Delta s_1)} \quad \text{Eq. 3-25}$$

If $a = b = 1$

$$f_1(\theta_{i_n}) = \frac{(1-\theta_{i_n})^{\Delta s_1}}{B(1, 1 + \Delta s_1)} \quad \text{Eq. 3-26}$$

- 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 because of 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 visits to node i_n if no new testing data are applied. This piece of information should be used as our only knowledge about the 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} .

$$f_1(\theta'_{i_n}) = C_1(1-\theta'_{i_n})^{\Delta s_1} f_0(\theta'_{i_n}) = \frac{\theta'_{i_n}^{a-1}(1-\theta'_{i_n})^{b+\Delta s_1-1}}{B(a, b + \Delta s_1)} \quad \text{Eq. 3-27}$$

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

$$f_1(\theta'_{i_n}) = \frac{(1-\theta'_{i_n})^{\Delta s_1}}{B(1, 1 + \Delta s_1)} \quad \text{Eq. 3-28}$$

The physics behind it is totally different though the formula is exactly the same as the result of Δs_1 failure-free test.

- Bayesian θ_{i_n} after Δs_2 further failure-free tests on node i_n :

Assume the distribution of θ_{i_n} becomes $f_1(\theta_{i_n})$. The underlying unreliability of node i_n has not changed and can be updated as follows:

$$f_2(\theta_{i_n}) = C_2 (1 - \theta_{i_n})^{\Delta s_2} f_0(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + \Delta s_1 + \Delta s_2 - 1}}{B(a, b + \Delta s_1 + \Delta s_2)} = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + s_2 - 1}}{B(a, b + s_2)} \quad \text{Eq. 3-29}$$

If $a = b = 1$

$$f_2(\theta_{i_n}) = \frac{(1 - \theta_{i_n})^{s_2}}{B(1, 1 + s_2)} \quad \text{Eq. 3-30}$$

- 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 $f_1(\theta_{i_n})$.

Assume that 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 that 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. Because 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.

$$f_2(\theta'_{i_n}) = C_2 (1 - \theta'_{i_n})^{\Delta s_1 + \Delta s_2} f_0(\theta'_{i_n}) = \frac{\theta'_{i_n}^{a-1} (1 - \theta'_{i_n})^{b + \Delta s_1 + \Delta s_2 - 1}}{B(a, b + \Delta s_1 + \Delta s_2)} = \frac{\theta'_{i_n}^{a-1} (1 - \theta'_{i_n})^{b + s_2 - 1}}{B(a, b + s_2)} \quad \text{Eq. 3-31}$$

If $a = b = 1$

$$f_2(\theta'_{i_n}) = \frac{(1 - \theta'_{i_n})^{s_2}}{B(1, 1 + s_2)} \quad \text{Eq. 3-32}$$

- 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:

$$f_k(\theta_{i_n}) = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + \Delta s_1 + \Delta s_2 + \dots + \Delta s_k - 1}}{B(a, b + \Delta s_1 + \Delta s_2 + \dots + \Delta s_k)} = \frac{\theta_{i_n}^{a-1} (1 - \theta_{i_n})^{b + s_k - 1}}{B(a, b + s_k)} \quad \text{Eq. 3-33}$$

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 $a = b = 1$

$$f_k(\theta_{i_n}) = \frac{(1 - \theta_{i_n})^{s_k}}{B(1, 1 + s_k)} \quad \text{Eq. 3-34}$$

The average of θ_{i_n} is:

$$\bar{\theta}_{i_n} = \int_0^1 \theta_{i_n} f_k(\theta_{i_n}) d\theta_{i_n} = \frac{a}{a + b + s_k} \quad \text{Eq. 3-35}$$

If $a = b = 1$

$$\bar{\theta}_{i_n} = \int_0^1 \theta_{i_n} f_k(\theta_{i_n}) d\theta_{i_n} = \frac{1}{2 + s_k} \quad \text{Eq. 3-36}$$

3.4.3.4. Nodal Coverage Based Reliability Estimation Method Conclusion

Nodal coverage reliability estimation approach is more approximate than flowpath coverage reliability estimation. But by looking at this method more carefully, we can observe several advantages to it because of its simple nature.

First, it helps us to achieve a balanced test. By examining the coverage status of each node, the un-tested nodes are picked out and input data is created manually to cover them. As shown in our demonstration work, in doing this, errors on those scarcely visited nodes are detected much sooner than they would have been by using simple random input data. Also, this process can help us to detect those nodes that can never be reached and should be removed from the program. This cannot be achieved by the more precise flowpath coverage-based approach that will be discussed later. There are flowpaths existing in the program because of the usefulness of all the nodes on them, but can never be reached by any input data set. Though it is feasible to find the surplus components of the software by examining nodal coverage status, it is impossible in the flowpath coverage approach. Furthermore it is not practical to examine the huge number of flowpaths within a typical program and pick certain input data to reach certain flowpaths, even for a testable case studied in this project.

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

Third, 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. Although 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 identify those input data that lead to a particular flowpath.

In summary, nodal coverage examination is very important. It helps to achieve a balanced test before any of the estimations is done. Some errors are more readily detected and surplus components of the software are more easily captured. Under the situation of relatively complicated software, nodal coverage estimation would be the substitute solution for the more precise path coverage estimation approach, discussed later in this report.

3.4.4. Refined Feasible Flowpath Coverage Reliability Estimation

The first reliability estimation design in this project is based on flowpath coverage, not on nodal coverage, as we described in previous section. That was the result of the completeness criteria, both in terms of testing and reliability estimation, required by the safety-critical nature of the software that we are working on. When we face the problem of feasibility, we relaxed our goal from a flowpath coverage based to a node coverage based method. After completing the nodal coverage based reliability estimation methodology demonstration, we wonder if we have taken advantage of all the information that we have obtained through the testing processes and if we can possibly acquire more constructive feedback to the testing process. Shall we stop the testing process right after we have achieved a complete nodal coverage on the program? The answer is no. Given that we have recorded all the tested flowpaths, we should be able to do some analysis based on these flowpaths. The stopping rule should be constructed in terms of flowpaths rather than nodes. The most fundamental questions are the percentage of flowpaths that have been tested, how reliable the tested flowpaths are, how reliable the untested flowpaths are, and the odds that they are going to be executed upon next execution, etc. All these questions are addressed in this refined feasible flowpath coverage reliability estimation methodology and exemplified in the sample software presented in the next chapter of this thesis, which has proved the practicality of the methodology.

3.4.4.1. Method General

In the refined feasible flowpath coverage reliability estimation approach, flowpaths of the software in question are divided into two groups, the tested flowpaths and the untested flowpaths. The overall unreliability of the software is estimated by the weighed average of the 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 every flowpath in this group. The unreliability of each tested flowpath is estimated using the testing results of that flowpath via the 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 and assume they are at the same unreliability level if no modification is made to either part of the software because these two parts are designed and programmed by the same software developing team under the same environment. The weight given to a flowpath or a group of flowpaths during the average calculation mentioned above is proportional to the probability that the 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, it is a different story. There is no theoretical or experimental method that we can use to calculate or measure how many flowpaths that have not been visited and which of them will be visited more often than the others. Fortunately we do not need to the relative visiting probability of each of the unvisited flowpaths because only a group-unreliability of the untested flowpaths can be deduced from the information that we obtained of the tested flowpaths. But we do need to know the chance of visiting one of them upon next execution of the program, which is anticipated by experimental curve fitting.

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

3.4.4.2. Method Details

3.4.4.2.1. Assumptions

- 1) Testing is representative of actual use.
- 2) Faults are of the same severity.
- 3) Detected faults are fixed immediately with certainty.
- 4) Visits to the same flowpath are statistically independent Bernoulli trials.
- 5) If no modification has been made to the tested flowpaths, on average, 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 an equal chance of being detected during any visit to the flowpath.

The first assumption is the same as that which 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. These 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, the issue is different visits to a flowpath. We take advantage of this assumption when updating reliability of a tested flowpath based on the testing results.

The fifth assumption says that if we have only tested some flowpaths in the program and not corrected any of the identified errors, then 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 of the program when they were developed. They are developed by the same group of people in the same 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 our work, we have divided all software errors into two categories from the testing point of view. Type I errors are always detected during the first time the flowpaths are visited. Type II errors have the same opportunity of being identified during any visit to those flowpaths. The error belongs to a flowpath or to a group of flowpaths. Because an error has to be within one node in order to belong to one or several flowpaths, it is important to note that an error does not necessarily cause all the flowpaths that contain this node to be at fault. 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.

Until now, we have mentioned completeness in this report several times. Can we actually achieve 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 complete flowpath coverage were performed on a piece of software, that is, if every possible flowpath were tested, it would be still possible that an undetected error would remain on one of the tested flowpaths. Thus, in this study, we do not assume that a flowpath is free of error regardless of whether it has been tested one time or one hundred times. Otherwise, it is definitely true that the flowpath that has been tested one hundred times is much more likely to be error free than the flowpath that has been tested only once. If we compare our approach on the possibility of an error existing with the traditional approaches, ours is a more conservative one. Also, as can be seen in later sections, the probability of an error existing in a flowpath decreases very quickly as the number of tests performed on this flowpath increases.

3.4.4.2.2. The General Formula for Flowpath Coverage Based Reliability Estimation

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:

$$\theta_p = P_{visit,T} \bar{\theta}_{p,T} + P_{visit,U} \bar{\theta}_{p,U} \quad \text{Eq. 3-37}$$

where

- θ_p is the software unreliability estimated by feasible flowpath coverage based reliability estimation approach
- $P_{visit,T}$ is the probability that one of the tested flowpaths is visited on next execution
- $P_{visit,U}$ is the probability that one of the untested flowpaths is visited on next execution
- $\bar{\theta}_{p,T}$ is the mean unreliability of the tested flowpaths
- $\bar{\theta}_{p,U}$ is the mean unreliability of the untested flowpaths

The mean unreliability of the tested flowpaths is defined as the probability that an error will be activated if one of the tested flowpaths 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 that an error exists that matters in this study.

Before we finish 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. However, we may never reach some of them because of the relationships among input data. Thus, when talking about complete path coverage testing, instead of saying all the paths should be tested, a more correct goal is to test every possible path throughout the flowgraph. There is not any analytical solution for the number of feasible flowpath yet. Therefore, we instead provide two strategies to estimate the total number of possible flowpaths and the probability that one of the tested flowpaths is going to be visited based on the experimental results obtained through the testing process.

3.4.4.2.3. Estimate the Average Unreliability of the Tested Flowpaths

This estimation is carried out in exactly the same way as is done to estimate the overall software reliability in the nodal coverage based reliability estimation approach. Every flowpath in this group is tested as least once, so that we can take its visiting frequency during testing as its relative visiting frequency within the group under a real operating environment. The estimation of unreliability of every tested flowpath depends 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:

$$\theta_{p,T} = \frac{\sum_{i_p=1}^{N_{p,T}} P_{visit,i_p} \hat{\theta}_{i_p}}{\sum_{i_p=1}^{N_{p,T}} P_{visit,i_p}} \quad \text{Eq. 3-38}$$

where

- $\theta_{p,T}$ is the average unreliability of the tested flowpaths,
- $N_{p,T}$ is the total number of tested flowpaths in the program,
- P_{visit,i_p} is the conditional probability of flowpath i_p being visited during real operation given that one of the tested flowpaths is visited,

$\hat{\theta}_{i_p}$ is the unreliability of flowpath i_p , estimated by Bayesian updating method.

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

$$P_{visit,i_p} = \frac{t_{visit,i_p}}{\sum_{i_p=1}^{N_{p,T}} t_{visit,i_p}} \quad \text{Eq. 3-39}$$

where

t_{visit,i_p} is the number of visits to flowpath i_p .

3.4.4.2.3.1. 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 because of a software failure caused by a fault. On the other hand, not all problems experienced arise from the classical software-programming bug that causes a failure. More often than not, a customer calls experiencing difficulties because of poor procedures, unclear documentation, poor user interfaces, etc.

So what is the definition of defect that 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 that we found during our testing process in a way such that the reliability of the software can be more precisely and easily estimated. In order to make this clear, we define a failure in the 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 actual plain language written design document. Once failure is defined, a defect is the cause in the software product that triggers a failure. In this report, the terms, error, fault and defect are 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 result 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 be removed in the case of safety-critical software, as used in nuclear power plants.

Now comes to the question of how we classify the problems experienced during the 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. Should an error exist within a flowpath, for a 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 for a 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 of all the others. There is one point that needs to be emphasized, i.e., both types of errors are associated with particular 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 because all nodes should have passed some level of modular testing. Here, both types of defects are believed to relate to some data flow that bypasses their control flows — flowpaths. For

the type I defects, almost all the input data, which lead to the flowpath where the error occurs, can trigger that error. For the type II defects, only part of the input data where leads 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 depend 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. Because each input data sampling process is independent of all the others and there are only two possible results, the input data belong either to the error-triggering group or to the non-error triggering group. Also, if the input data can lead to the execution of a flowpath where a type II error occurs, then each process is an independent Bernoulli process. An obvious example of a type II error is a boundary-condition error, which is only activated when a particular 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 boundary type errors.

In order to incorporate this two-error-type scenario in our reliability estimation process, two important probability values should be available. The first one is the probability that an error chosen is of type I if it is randomly chosen from a pool of errors from the target software or similar software systems. In other words, the first item that 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, is then $p_{II} = 1 - p_I$. The second item that 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 the flowpath in question has a type II error since each input data set is chosen independently and in the same manner. Let us denote this probability as $p_{f,II}$. For a 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 that there is a type I error on that flowpath, is $p_{f,I}^1 = 1$. Because it is required that all the errors be removed upon detection, we will not again

encounter this error during further visits to the same flowpath, which means $p_{f,I}^{+} = 0$. It is true that p_I , p_{II} , and $p_{f,II}$ are different for different programs, even for different flowpaths in the same software system. But because 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 an average sense. That is 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 the Bayesian framework. With the distributions on hand, the mean values or median values could be applied in order to estimate reliabilities / unreliabilities of the flowpaths, hence to estimate the reliability of the software.

3.4.4.2.3.2. Update Probability Distributions Associated with The Two Error Types

In this section, a brief review on the Bayesian updating method is presented, followed by 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 use of the Bayesian method, there have been several classical statistical approaches, such as point and interval estimation to estimate the distribution parameters. These approaches assume that the parameters of interest are unknown constants 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 the values of parameters in these approaches require a large amount of data. The Bayesian framework addresses the estimation problem from another point of view. In this case, the unknown parameters of a distribution are assumed (or modeled) to be random variables. In this way, the 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.

Bayes' theorem is stated 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:

$$P(E_i | A) = \frac{P(A | E_i)P(E_i)}{P(A)} = \frac{P(A | E_i)P(E_i)}{\sum_{j=1}^n P(A | E_j)P(E_j)} \quad \text{Eq. 3-40}$$

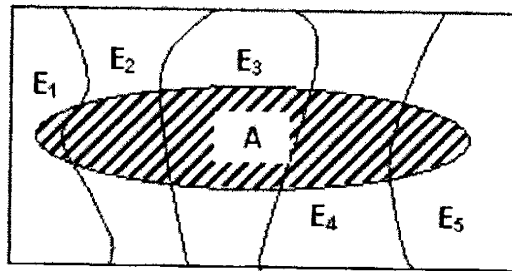


Figure 3-9 Venn diagram with events A and E_1, E_2, \dots, E_n

The 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 a 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 concerning 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:

$$P(\Theta = \theta_i | \varepsilon) = \frac{P(\varepsilon | \Theta = \theta_i)P(\Theta = \theta_i)}{\sum_{j=1}^n P(\varepsilon | \Theta = \theta_j)P(\Theta = \theta_j)}, \quad i = 1, 2, \dots, n \quad \text{Eq. 3-41}$$

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:

$$P''(\Theta = \theta_i) = \frac{P(\varepsilon | \Theta = \theta_i)P'(\Theta = \theta_i)}{\sum_{j=1}^n P(\varepsilon | \Theta = \theta_j)P'(\Theta = \theta_j)} \quad \text{Eq. 3-42}$$

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

$$\hat{\theta}'' = E(\Theta | \varepsilon) = \sum_{j=1}^n \theta_j P''(\Theta = \theta_j) \quad \text{Eq. 3-43}$$

The continuous formulas for Bayesian method are:

$$f''(\theta) = \frac{P(\varepsilon | \theta) f'(\theta)}{\int_{-\infty}^{\infty} P(\varepsilon | \theta) f'(\theta) d\theta} \quad \text{Eq. 3-44}$$

$$\hat{\theta}'' = \int_{-\infty}^{\infty} \theta f''(\theta) d\theta \quad \text{Eq. 3-45}$$

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

If we are able to distinguish the type I and type II errors, that is, if we can divide all the encountered errors into the two error groups, then 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 causes behind the error should be examined and 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 that we need to use to update p_I , p_{II} and $p_{f,II}$. The steps are as follows.

Let's assume we only use error data from the testing results of the program under test 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 default uniform probability distribution:

$$f_0(p_I) = 1 \quad \text{Eq. 3-46}$$

Alternatively, we can use a prior reflecting typical software quality obtained using modern development methods. If the first encountered error is of type I, then $f_0(p_I)$ is updated as:

$$f_1(p_I) = C_1 p_I f_0(p_I) \quad \text{Eq. 3-47}$$

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:

$$\begin{aligned} \int_0^1 f_1(p_I) dp_I &= \int_0^1 C_1 p_I f_0(p_I) dp_I = 1 \\ \Rightarrow C_1 &= \frac{1}{\int_0^1 p_I f_0(p_I) dp_I} \end{aligned} \quad \text{Eq. 3-48}$$

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

$$f_2(p_I) = C'_2 (1 - p_I) f_1(p_I) = C_2 p_I (1 - p_I) f_0(p_I) \quad \text{Eq. 3-49}$$

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

In general,

$$f_s(p_I) = \begin{cases} C_s p_I f_{s-1}(p_I) & C_s = \frac{1}{\int_0^1 p_I f_{s-1}(p_I) dp_I} \quad \text{type I error} \\ C_s (1 - p_I) f_{s-1}(p_I) & C_s = \frac{1}{\int_0^1 (1 - p_I) f_{s-1}(p_I) dp_I} \quad \text{type II error} \end{cases} \quad \text{Eq. 3-50}$$

Note that we have been taking $[0, 1]$ as the integral range of p_I . This is because p_I is a probability, which implies that 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:

$$\hat{p}_I = \int p_I f_s(p_I) dp_I, \quad \text{Eq. 3-51}$$

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

$$\hat{p}_{II} = 1 - \hat{p}_I \quad \text{Eq. 3-52}$$

Now we need to update the 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 if they are within the same program. But because only the average effect matters to us, the average probability can be taken as a constant and updated based on the Bayesian method. We 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 default uniform probability distribution:

$$f_0(p_{f,II}) = 1 \quad \text{Eq. 3-53}$$

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

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

$$C_1 = \frac{1}{\int_0^1 (1 - p_{f,II})^3 p_{f,II} f_0(p_{f,II}) dp_{f,II}}$$
Eq. 3-54

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

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

$$C_2 = \frac{1}{\int_0^1 (1 - p_{f,II})^4 p_{f,II}^2 f_0(p_{f,II}) dp_{f,II}}$$
Eq. 3-55

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} time as:

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

$$C_s = \frac{1}{\int_0^1 (1 - p_{f,II})^{k_s-1} p_{f,II} f_{s-1}(p_{f,II}) dp_{f,II}}$$
Eq. 3-56

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:

$$\hat{p}_{f,II} = \int p_{f,II} f_{N_{e,II}}(p_{f,II}) dp_{f,II}$$

Eq. 3-57

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

What should we do if we can not separate the two groups of errors? This situation is not very unusual because 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. It is to update the joint probability distribution function of p_I and $p_{f,II}$, considering that they are independent of each other. It is obvious that the fraction of type I error has nothing to do with the probability of detecting a type II errors during a single visit to a flowpath that has a type II error. Now, the only error data that we need is what number of visits it takes to detect an error.

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

$$f_0(p_I, p_{f,II}) = f_0(p_I) f_0(p_{f,II}), \quad 0 \leq p_I \leq 1, \quad 0 \leq p_{f,II} \leq 1$$

Eq. 3-58

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

$$f_0(p_I) = f_0(p_{f,II}) = f_0(p_I, p_{f,II})$$

Eq. 3-59

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

$$f_1(p_I, p_{f,II}) = C_1(1 - p_I)(1 - p_{f,II})p_{f,II}f_0(p_I, p_{f,II}) \quad \text{Eq. 3-60}$$

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

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 equal to zero because of 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:

$$\begin{aligned} f_2(p_I, p_{f,II}) &= C'_2[p_I + (1 - p_I)p_{f,II}]f_1(p_I, p_{f,II}) \\ f_2(p_I, p_{f,II}) &= C_2[p_I + (1 - p_I)p_{f,II}](1 - p_I)(1 - p_{f,II})p_{f,II}f_0(p_I, p_{f,II}) \end{aligned} \quad \text{Eq. 3-61}$$

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

In general, if the 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:

$$\begin{aligned} f_s(p_I, p_{f,II}) &= C_s[p_I + (1 - p_I)p_{f,II}]f_{s-1}(p_I, p_{f,II}) \\ C_s &= \frac{1}{\iint_{p_I, p_{f,II}} f_s(p_I, p_{f,II})dp_I dp_{f,II}} \end{aligned} \quad \text{Eq. 3-62}$$

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. 3-63

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. 3-64

Also 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. 3-65

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

3.4.4.2.3.3. 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 information we can use to estimate or update unreliability of flowpath i_p , θ_{i_p} ? It is in the same form as in the nodal coverage based approach. For every tested flowpath, it is number of failure free tests t_{i_p} that have been performed on the flowpath because it is required that all known faults in the software program be removed.

Now, let's estimate the unreliability of a flowpath, θ , given that 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,

$$\theta = p_e \times p_f \quad \text{Eq. 3-66}$$

where

p_e is the probability that there is an error on the flowpath

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

From above formula, it seems that the probability that there is more than one error on one flowpath is ignored. This is not the case. When 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 resulting unreliability estimates so as to indicate the testing results correctly. 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 times 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 the 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.

$$f_1(p_e) = P(p_e | \bar{F}) = \frac{P(p_e, \bar{F})}{P(\bar{F})} = \frac{P(\bar{F} | p_e) f_0(p_e)}{\int P(\bar{F} | p_e) f_0(p_e) dp_e}, \quad 0 \leq p_e \leq 1 \quad \text{Eq. 3-67}$$

$$f_1(p_e) = \frac{[P(\bar{F}, E | p_e) + P(\bar{F}, \bar{E} | p_e)] f_0(p_e)}{\int [P(\bar{F}, E | p_e) + P(\bar{F}, \bar{E} | p_e)] f_0(p_e) dp_e} \quad \text{Eq. 3-68}$$

$$f_1(p_e) = \frac{[P(\bar{F} | E, p_e) P(E | p_e) + P(\bar{F} | \bar{E}, p_e) P(\bar{E} | p_e)] f_0(p_e)}{\int [P(\bar{F} | E, p_e) P(E | p_e) + P(\bar{F} | \bar{E}, p_e) P(\bar{E} | p_e)] f_0(p_e) dp_e} \quad \text{Eq. 3-69}$$

where

\bar{F} is the flowpath has been tested once without any error,

E is there is an error on the flowpath,

\bar{E} is there is no error on the flowpath,

p_e is the probability that there is an error on the flowpath,

$f_0(p_e)$ is the prior probability density function of p_e ,

$f_1(p_e)$ is the 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:

$$P(\bar{F} | E, p_e) = P(\bar{F} | E_I, E, p_e)P(E_I | E, p_e) + P(\bar{F} | E_{II}, E, p_e)P(E_{II} | E, p_e) \quad \text{Eq. 3-70}$$

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

The quantity $P(E_I | E, p_e)$ is the probability that a randomly 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, $P(E_I | E, p_e) = p_I$.

The quantity $P(\bar{F} | E_{II}, E, p_e)$ is the conditional probability that we do not encounter a failure during a test given that there is a type II and only a type II error on the flowpath. According to the definition of $p_{f,II}$, which is 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 randomly selected error belongs to type II, or, the fraction of type II error amount all errors. Because 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:

$$P(\bar{F} | E, p_e) = (1 - p_{f,II})(1 - p_I)$$

Eq. 3-71

It is apparent that if there is no error on the flowpath, we will not encounter any error during any test. Thus, we have the result:

$$P(\bar{F} | \bar{E}, p_e) = 1 \quad \text{Eq. 3-72}$$

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

$$\begin{aligned} f_1(p_e) &= C_1 [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)] f_0(p_e) \\ C_1 &= \int_0^1 [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)] f_0(p_e) dp_e, \quad 0 \leq p_e \leq 1 \end{aligned} \quad \text{Eq. 3-73}$$

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

$$\begin{aligned} f_2(p_e) &= C_2' [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)] f_1(p_e) \\ f_2(p_e) &= C_2 [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)]^2 f_0(p_e), \quad 0 \leq p_e \leq 1 \\ C_2 &= \int_0^1 [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)]^2 f_0(p_e) dp_e \end{aligned} \quad \text{Eq. 3-74}$$

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

$$\begin{aligned} f_t(p_e) &= C_t [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)]^t f_0(p_e) \\ C_t &= \int_0^1 [p_e(1 - p_l)(1 - p_{f,II}) + (1 - p_e)]^t f_0(p_e) dp_e, \quad 0 \leq p_e \leq 1 \end{aligned} \quad \text{Eq. 3-75}$$

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

$$\hat{p}_e = C_t \int [p_e(1-p_t)(1-p_{f,II}) + (1-p_e)]^t p_e f_0(p_e) dp_e \quad \text{Eq. 3-76}$$

$$C_t = \int [p_e(1-p_t)(1-p_{f,II}) + (1-p_e)]^t f_0(p_e) dp_e$$

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

Sometimes, the above discussed Bayesian updating technique can be very difficult to implement depending on the form of prior distribution $f_0(p_e)$. Therefore, we want to introduce a simplified method of estimating \hat{p}_e . Assume that p_e is a value and not as in the Bayesian updating, a distribution. After we have performed t tests on a flowpath without any error, p_e can be written as:

$$p'_e = P(E | \bar{F}') = \frac{P(E, \bar{F}')}{P(\bar{F}')} = \frac{P(\bar{F}' | E)P(E)}{P(\bar{F}' | E)P(E) + P(\bar{F}' | \bar{E})P(\bar{E})} \quad \text{Eq. 3-77}$$

where

- \bar{F}' is the flowpath has been tested t times without any error,
- E is there is an error on the flowpath,
- \bar{E} is there is no error on the flowpath,
- p'_e is the probability that there is an error on the flowpath after t error free test on the flowpath.

As discussed above, the following relationships are obtained:

$$\begin{aligned}
 P(\bar{F}' | E) &= (1 - p_l)(1 - p_{f,II})' \\
 P(\bar{F}' | \bar{E}) &= 1 \\
 P(E) &= p_e^0 \\
 P(\bar{E}) &= 1 - p_e^0
 \end{aligned}
 \tag{Eq.3-78}$$

where, p_e^0 is the prior value of the probability that there is an error on the flowpath. 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' is not necessarily the same as \hat{p}_e . To sum up, this simpler method to estimate p_e is:

$$p_e' = \frac{(1 - \hat{p}_l)(1 - \hat{p}_{f,II})' p_e^0}{(1 - \hat{p}_l)(1 - \hat{p}_{f,II})' p_e^0 + (1 - p_e^0)}
 \tag{Eq. 3-79}$$

For both methods, the estimators, \hat{p}_l and $\hat{p}_{f,II}$ obtained from the error data of the same program or similar programs are substituted into their respective formulas to obtain the probability of error existing in each tested flowpath.

In order to estimate the unreliability of a tested flowpath, θ , besides p_e , which can be solved either by Eq. 3-76, or Eq. 3-79, we have to solve p_f , the probability that a failure will be encountered upon next execution on the flowpath given that there is an error on the flowpath according to Eq. 3-66. Here another assumption is made. Unlike p_e , which is unique to every tested flowpath, an average value is used for p_f for every flowpath. This is because we have sparse error data from the software that we are testing and this assumption 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 a 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:

$$p_f = P(F | E_I)p'_I + P(F | E_{II})p'_{II} \quad \text{Eq. 3-80}$$

where

- p_f is the probability that a failure is encountered upon next execution given there is an error on the flowpath,
- $P(F | E_I)$ is the probability that a failure is encountered upon next execution given there is a type I error on the flowpath,
- p'_I is the probability that an error is of type I, given it has been tested t times failure free, $t \geq 1$,
- $P(F | E_{II})$ is the probability that a failure is encountered upon next execution given there is a type II error on the flowpath,
- p'_{II} is 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 with which we are dealing are those flowpaths that have been tested at least once without any failure. Where there is a type I error on any of these flowpaths, it should have been encountered during the first visit according to the definition of type I error. This is to say $p'_I = 0$. Thus, there is no need to calculate $P(F | E_I)$ for the purpose of estimating p_f . Also, because $p'_I + p'_{II} = 1$, and because there are only two types of errors, $p'_{II} = 1 - p'_I = 1$. The last term is $P(F | E_{II})$, which is 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 Eq. 3-80, we get:

$$P_f = P_{f,II}$$

Eq. 3-81

3.4.4.2.3.4. Summary — Estimate Unreliability of Tested Flowpaths

According to Eq. 3-38, the unreliability of the tested part of the software is a weighted average of the 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 Eq. 3-66, 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 estimated by Eq. 3-76, assuming continuous distribution for p_e or Eq. 3-79, only caring about the average value of p_e . The probability that a failure is encountered during the 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 the 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 flowpaths can have 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 because of 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. 3-50, Eq. 3-51, Eq. 3-52, Eq. 3-56 and Eq. 3-57, which is easier in terms of calculation. If the tester cannot distinguish type I errors from type II errors, a joint probability distribution of p_I and $p_{II,f}$ can be carried out and estimations obtained according to Eq. 3-63, Eq. 3-65.

3.4.4.2.4. 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 possible flowpaths are tested. Furthermore, we do not even have a method to count how many possible flowpaths that exist in a program. (This topic is discussed in detail in the next section). The only information that we obtain after the testing is that from the tested flowpaths. Our task now is to estimate the unreliability of the untested flowpaths based on the information we have obtained from the tested flowpaths. In order 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 that are shared by them and bridge the differences. Thus, in this section, we start with the characteristics of the tested and untested parts and continue with how to deal with them in order to obtain average unreliability estimation for the untested flowpaths from the testing information that we have obtained from the tested flowpaths.

3.4.4.2.4.1. Similarities And Differences between Tested and Untested Flowpaths

First, the tested and untested flowpaths belong to the same software program. They share many common nodes. They would share all the nodes if complete nodal coverage testing has been performed on the underlying software. Both are designed and programmed by the same group of people under the same development environment. It is reasonable to assume that they are at the same reliability level if neither has been tested or modified. That level is the average reliability of the software before testing. Now suppose we test some parts of the software, without any correction to any of the detected errors? Are they still the same? The answer is yes. It is 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 so that at least some of them are successfully and perfectly corrected, does the reliability increase. 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 are able to update software reliability based upon the pure testing results, it is not because the underlying reliability has changed. Rather it is because we have more knowledge about the software and can obtain a more precise estimation of its reliability. Testing provides information of the tested flowpaths and permits error removal from the tested flowpaths. Thus, we can say that 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 correction 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 test and modify the program at the same time. In this case, we test the software and always perfectly correct any error that is detected during testing. We increase the reliability of the tested flowpaths if we only care about the average reliability of the tested and untested flowpaths. And we here introduce a difference between the tested and the untested flowpaths in terms of reliability by modifying the tested flowpaths.

Now, we believe that the reliability of the tested flowpaths is higher than the reliability of the untested flowpaths. But is the untested part as unreliable as the software before any test and modification was performed on the software? Does the modification of the tested flowpaths improve the quality of the untested flowpaths? Yes, because they share many nodes. It is possible that if an error on a node is corrected because of testing of a tested flowpath, some potential failure observations from the untested flowpath are removed too. In other words, because of the error being found on the same node shared by different flowpaths belonging to both the tested flowpaths and to the untested flowpaths, we will observe fewer failures on the untested flowpaths than if we do not make any modification to 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 because we cannot manage complete flowpath coverage on the underlying software, the tested flowpaths are modified while the untested flowpaths are not. This causes the difference to develop between these two parts of the software 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 the initial reliability level of the software without any modification because of the common errors on common nodes shared by both parts of the software. This is the beneficial effect on the untested flowpaths derived from modifying the tested flowpaths.

3.4.4.2.4.2. Unreliability of Untested Flowpaths

There are two versions of reliability because of 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 version of reliability is the one in which we are interested. 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.

One error may cause several failures if we do not perfectly correct it immediately. The reason is that there is normally more than one flowpath sharing the faulty node and this node 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 3-10, 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 because of the error on node 5. From the 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 most⁷.

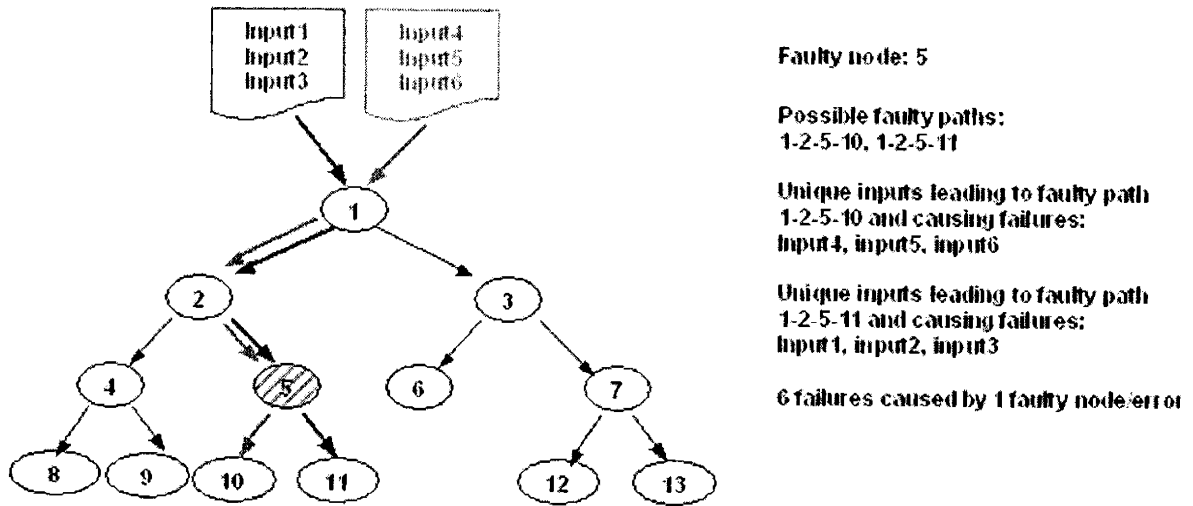


Figure 3-10 Six Failure Observations Because of One Error

Now, let us define two terminologies, the observed reliability and the basic reliability. The reliability estimated using the number of errors encountered during the testing and perfect correcting process is termed the observed reliability because this is what we see when we test the tested flowpaths and perfect correct any detected errors. The reliability estimated from the number of errors encountered during a pure testing (test without error removal) process is terms as the basic reliability. If none of the errors are shared by more than one test case, the basic reliability is equal to the observed reliability. The observed reliability is always equal to or greater than the basic reliability. We term this phenomenon a shielding effect because the perfect correction of a detected error on a tested flowpath shields our further observations of the same error from other tests. The shielding effect is measured by the ratio of the observed reliability to the basic reliability, R_O / R_B , which is always greater than or equal to one.

Note that neither the observed reliability nor the basic reliability estimated from the number of errors encountered during testing of the tested flowpaths can reflect the

⁷ Because 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. Six is the maximum number of failures that could be caused by node 5.

true reliability value of the tested flowpaths. Both estimates are made upon the fact that some failures are observed during a certain number of tests. Because of the perfect-error-removal requirement, no error should be observed by applying all the existing input data to the software under question. The reliability estimated for the tested flowpaths should always be higher than both the observed reliability estimate and the basic reliability estimate. The observed and basic reliability of the tested flowpaths that are estimated from the testing information obtained from the tested flowpaths are not used to estimate reliability of the same part of software.

Then why do we want to estimate the observed and basic reliability of the tested flowpaths? We want to deduce reliability of the untested and unmodified flowpaths from these two reliability estimates. As mentioned above, there are lots of similarities between these two parts. Thus the error data from the tested portion should give us some hint about the untested portion.

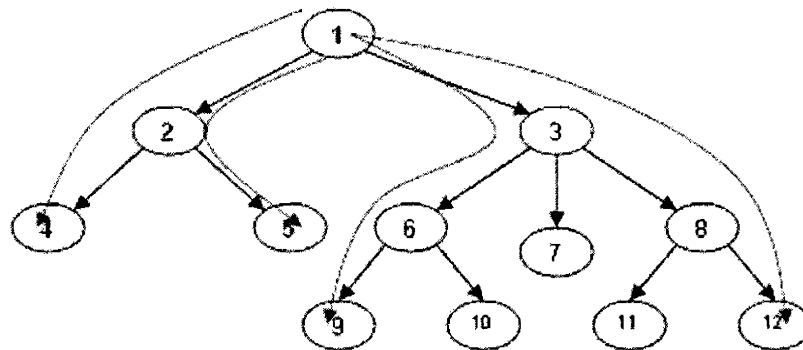


Figure 3-11 Tested and Untested Flowpaths, An Example

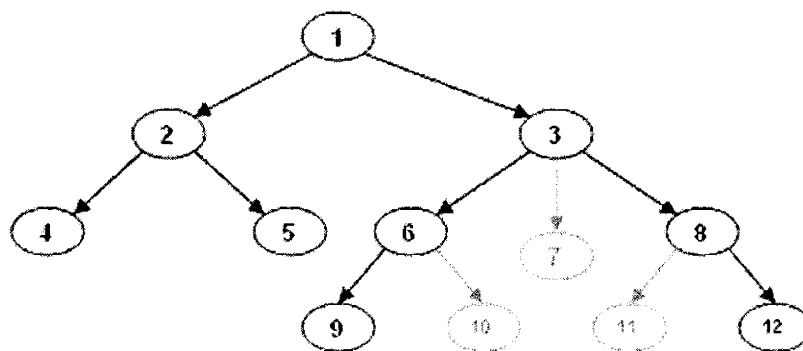


Figure 3-12 Tested Flowpaths, Part A

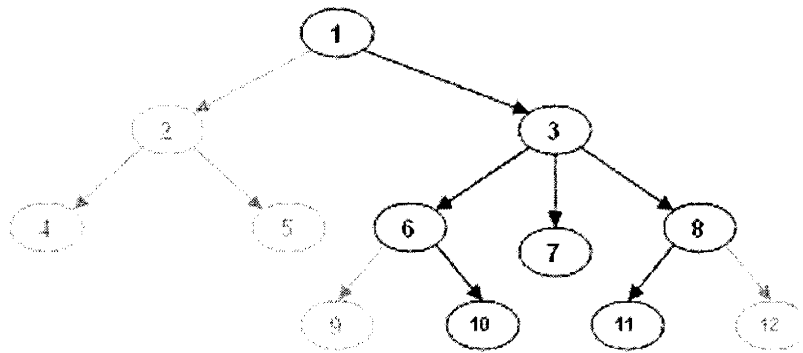


Figure 3-13 Untested Flowpaths, Part B

The first question we want to ask ourselves is whether the observed reliability or the basic reliability is the reliability value we want to estimate for the untested flowpaths. It is the basic reliability. At any testing stage, we always want to decide if we can stop testing and put the program into unlimited operation when no modification will be made in the future. This situation is the same as the pure testing process. It is the basic reliability level that will be experienced during this process. To estimate basic reliability of the untested flowpaths, our approach starts from the known, the observed reliability of the tested flowpaths, from where the observed reliability of the untested flowpaths is deduced. Then the relationship between basic reliability and observed reliability of the untested flowpaths (shielding effect within the untested flowpaths) is established by examining the relationship between the basic reliability and observed reliability of the tested flowpaths (shielding effect within the tested flowpaths).

Assume we can divide the program into two parts. For the example shown in Figure 3-11, four out of seven flowpaths have been tested. The software can be divided into two parts as in Figure 3-12 and Figure 3-13. If these two parts were independent, the reliability level of the untested flowpaths under unlimited operating use would be the same as the basic reliability estimated for the tested flowpaths provided the similarities these two parts are sharing. The problem is that the tested flowpaths and the untested flowpaths are not independent at all. In the example plotted in Figure 3-11, part A and part B share node 1, 3, 6 and 8. Through the shared nodes, modifications made to the tested flowpaths also benefit the untested flowpaths. Thus if we were to use the basic

reliability of the tested flowpaths assuming no modification have been made to estimate reliability of the untested flowpaths, we have underestimated the reliability, which makes our estimation more conservative. If we use a multiplication correction factor C_1 to denote this effect, we have

$$R_o(\text{untested}) = C_1 R_o(\text{tested}), \quad C_1 \geq 1 \quad \text{Eq. 3-82}$$

The true reliability is always higher or equal to our reliability estimation value. Considering the safety-critical nature of the software in question, we can ignore C_1 , or assume $C_1 = 1$. By ignoring the benefit brought to the untested flowpaths by correcting the errors detected on the tested flowpaths, we obtain a more conservative reliability estimation result for the untested flowpaths. This is expressed as:

$$R_o(\text{untested}) \approx R_o(\text{tested}) \quad \text{Eq. 3-83}$$

Now let us estimate reliability of the untested flowpaths ignoring C_1 . The question becomes how to estimate the basic reliability of the untested flowpaths, $R_B(\text{untested})$, from the observed reliability of the untested flowpaths, $R_o(\text{untested})$. This is when the shielding effect comes into play. Again the shielding effect is the reduction of error observations as a result of prompt error removal. To account for the shielding effect, we define a correction multiplication factor similar to C_1 as:

$$C_{SE} = R_o / R_B, \quad C_{SE} \geq 1 \quad \text{Eq. 3-84}$$

This is a factor affect the reliability estimation in a different direction from C_1 . If we ignore C_{SE} , we will over-estimate reliability of the untested flowpaths. This is not acceptable for the safety-critical software in question.

Note that C_{SE} is not a constant throughout a program. It is a variable whose value associated directly with certain errors. If a node hosting any error is shared by more flowpaths and the flowpaths are the popular ones (more probable to be visited), it is more possible that the error will cause more failures if it is not removed immediately from the program. To put it in a simple way, such errors will bring bigger shielding effect to the program. If we denote the average shielding effect brought by the errors left in the untested flowpaths as $C_{SE}(untested)$, the basic reliability of the untested flowpaths of the untested flowpaths can be written as:

$$R_B(untested) = R_O(untested) / C_{SE} = C_1 R_O(tested) / C_{SE} \approx R_O(tested) / C_{SE} \quad \text{Eq. 3-85}$$

Because $R_O(tested)$ can be estimated directly from information obtained from the tested flowpaths, the only unknown becomes $C_{SE}(untested)$.

By intuition, $C_{SE}(untested)$ is much smaller than $C_{SE}(tested)$ because the errors have bigger shielding effect should also be easier to detect. Suppose the program has been tested many times and there is an error undetected, that error should be very little affect. Otherwise, we would have found it through previous testing.

Keep the above analysis in mind; we can associate a shielding effect value with each of the errors in given program. The earlier detected errors always have bigger shielding effect value associated with it than the later detected ones. The shielding effect factors of the detected errors can be plotted as a function of the time (the index of the flowpath under testing when the error is detected) when the errors are identified. Refer to Figure 3-14 for illustrations of such a plot. The vertical axis is the ratio between observed reliability and the basic reliability calculated for each of the detected errors, its shielding effect factor, R_O / R_B . The horizontal axis, n , is the indexes of the flowpaths under testing when the errors are detected. According to the discussion in the above paragraph, the curve is monotonously decreasing. As the more commonly shared faulty nodes are more easily to reach, they tend to be gotten rid of at earlier testing and modifying stages. As a result, the curve decreases dramatically as the number of total

tested flowpaths increases. To start, we suggest a function format between $R_O / R_B(n)$ and n as:

$$R_O / R_B = 1 + ae^{-bn} \quad \text{Eq. 3-86}$$

- R_O is the observed reliability
- R_B is the basic reliability
- n is the flowpath index when the error is detected
- a is the first function parameter
- b is the function parameter, indicating the decreasing speed of R_O / R_B as number of tested flowpaths increases

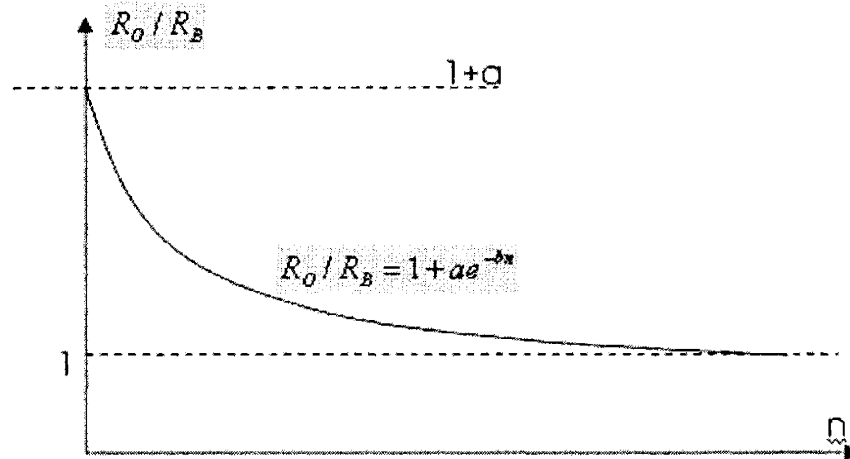


Figure 3-14 Ratio of Observed Reliability over Basic Reliability

Experiments on the software under consideration or similar programs should be performed to get approximation for a and b . This experiment can be performed after some number of tests. For each of the error detected and removed, first assume there is only one failure caused by this error plus there is no other failure out of the passed test cases. Calculate the reliability of the tested flowpaths according to the method described

earlier. This gives us the observed reliability associated with this error, $R_O(n)$. Then put back this error and test all the already passed test cases again. Presumably more than one failure will be observed because of the shielding effect. Use this testing result and recalculate reliability of the tested flowpaths. We get the basic reliability associated with this error, $R_B(n)$. Divide $R_O(n)$ by $R_B(n)$, we get the ratio we want. We do the same experiment for every error that has been found and estimate a and b . As shown in Figure 3-14, if a lot of tests have been done and most of the errors have been removed, the shielding effect is gone. C_{SE} of any of the errors left in the software, if there is any, can be approximated by one, which means no shielding effect exists with them. In this case, we have:

$$R_B(\text{untested}) = C_1 R_O(\text{tested}) / C_{SE}(\text{untested}) \approx R_O(\text{tested}) \quad \text{Eq. 3-87}$$

Then how to calculate the basic reliability and observed reliability associated with an error given all the information we need. Let's look at a simple example. If we have tested a program 20 times and have identified 8 unique flowpaths. There is one type II error that was detected on flowpath number 2 and has been removed. Refer to Table 3-2 for the example.

Flowpath Index i	1	2	3	4	5	6	7	8
No. of Visits t	5	3	4	3	2	1	1	1
Error Detected	0	1	0	0	0	0	0	0

Table 3-2 Example for Observed Reliability Estimation

From the above prompt error removal process, we can estimate the observed reliability as follows. For flowpath number 2, the probability that there is a type II error is unity, that is $Pe_2 = 1$. Its unreliability is estimated as:

$$\hat{\theta}_2 = Pe_2 \times p_{f,II} = p_{f,II} \quad \text{Eq. 3-88}$$

where, $p_{f,II}$ is the probability that a type II error is detected during a single visit to the hosting flowpath.

For all the other flowpaths, the unreliability can be estimated as:

$$\hat{\theta}_i = \frac{(1 - p_{f,II})^t p_{II} Pe^0}{(1 - p_{f,II})^t p_{II} Pe^0 + (1 - Pe^0)} p_{f,II}, \quad i \neq 2 \quad \text{Eq. 3-89}$$

where,

t is the number of visits to flowpath i during testing;

$p_{f,II}$ is the probability a type II error is detected during a single visit to the hosting flowpath.

p_{II} is the percentage of type II error among all errors.

Pe^0 is the average prior probability that there is an error on a single flowpath.

Then the observed unreliability of the 10 tested flowpaths is:

$$\hat{\theta} = \frac{\sum_{i=1}^{10} t_i \hat{\theta}_i}{\sum_{i=1}^{10} t_i} \quad \text{Eq. 3-90}$$

Now let's estimate the basic reliability of the 10 tested flowpaths. Assume we put back the removed error, repeat the 20 tests and observe the testing data as follows:

Flowpath Index i	1	2	3	4	5	6	7	8
No. of Visits t	5	3	4	3	2	1	1	1
Error Detected	2	1	1	1	0	0	0	0

Table 3-3 Example for Basic Reliability Estimation

To estimate the overall basic unreliability, we can estimate basic unreliability of node 1, 2, 3, 4, 5 according to Eq. 3-88 and estimate basic unreliability of the other nodes according to Eq. 3-89. The overall basic unreliability can be calculated according to Eq. 3-90. The basic/observed reliability is one minus the basic/observed unreliability.

If we were only to approach the point when the shielding effect disappears, we can apply a simpler version of the above experiment. Instead of calculating R_b and R_o for each error that was encountered, the total number of failures that was encountered when testing the error-embedded program is recorded. As discussed already, the later detected errors will cause fewer failures than the errors detected earlier. Again, we can plot the number of failures caused by each error as a function of the index of the flowpath that is under testing when the error was found. When we reach the point where one error only affects that single flowpath, we know that this is when the shielding effect disappears.

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

We have discussed how to estimate the reliability of the tested and untested flowpaths. But we should not forget one very important presumption, that is, the total number of executable flowpaths, N , is unknown. Without N , we do not know what percentage of the program has been tested. It is obvious that the tested and modified flowpaths are more reliable than the untested ones. And we can estimate reliability for both parts. What is also very important and we do not know is the chance that one of the

untested flowpaths will be executed upon next execution, $p_{visit,U}$ and what is the chance for one of the tested flowpaths $p_{visit,T}$. Apparently they have such relationship: $p_{visit,T} = 1 - p_{visit,U}$. Until now, we have not addressed this problem. Will this question be answered if we can estimate the total number of flowpaths?

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

3.4.4.2.5.1. 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 is a sequence of nodes and links on the flowgraph. It is the decision nodes that determine 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 that the program execution 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.

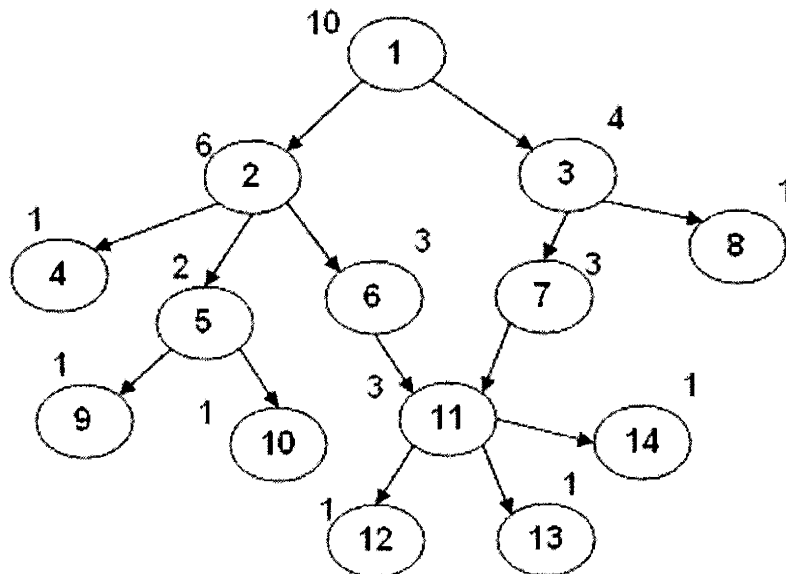


Figure 3-15 Calculate Number of Flowpaths Theoretically

Consider the flowgraph in Figure 3-15 as an example. The theoretical number of flowpaths in a program can be calculated using the 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. Node 4, 9, 10, 12, 13, 14, 8 are 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. Because 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 go 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. A value 1 is put into both node 9 and node 10's recorder because 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 3-4 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	$2 = 1 \text{ (node10)} + 1 \text{ (node11)}$
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 3-4 Illustration of Calculation of
Number of Flowpaths of Program Shown in Figure 3-15

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, because of special relationships among different input variables, some theoretically existing flowpaths may not be reached in practice. This has already been mentioned when we talked about the white box testing strategies. A simple example would be two adjacent if/else selection structures whose decision making conditions include duplicate variables. In such a case, some input data that lead to the if-block of the first if/else structure would not be able to reach either the if- or the else-block of the second if/else structure. Again, let us take the flowgraph in Figure 3-15 as an example. If the input data that leads the program to node 2 can only lead the program to node 12 but neither to node 13 nor 14 while the input data that leads the program to node 3 can only lead the program to nodes 13 and 14, but not to node 12. And assuming that all the other flowpaths are reachable, there are only 7 possible flowpaths. Flowpaths 1-2-6-11-13, 1-2-6-11-14 and 1-3-7-11-12 cannot be reached. In reality, there are many similar situations that make the theoretically calculated total number of flowpaths almost useless. Furthermore, performing a depth-first traversal with a real software control flowgraph is anything but an easy task, considering all the looping structures and multiple-use functions blocks, even though the software structure is very simple and clear as in the example shown in Figure 3-15.

Now, let us look at the probability that an untested flowpath is going to be executed during the next trial, $p_{visit,U}$. Note that $p_{visit,U} \neq \frac{N - N_T}{N}$ because different flowpaths have different probabilities of being accessed during operation. It is almost always true that after many tests have been performed on a program, the tested flowpaths always have a better chance to be visited on the next trial if the test data sampling process follows the same underlying distribution. This means that $p_{visit,U} \ll \frac{N - N_T}{N}$. But how small is the actual number? The total number of flowpaths in a program can be estimated from the program structure even though 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. In the following two sections, two experimental methods to estimate the total number of possible flowpaths in a program, as well as $p_{visit,T}$ and $p_{visit,U}$ are discussed.

3.4.4.2.5.2. First Experimental Solution

In this, and the following sections, two empirical solutions to the total number of flowpaths problem are provided. They appear naturally as the demonstration work, discussed in the next chapter, proceeds and generates a huge number of data points. Though they are derived from observation of the pattern obtained from experimental data, there are intuitive reasons that support them. Both are black box methods, meaning that in order to obtain 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 in order to obtain a satisfactory solution. 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, many ants are needed to map them all, account for the fact that several ants may end up traversing the same path. Furthermore, the solution will vary if the input data distribution

changes to some extent, depending on the size of the input data set. But as long as we are close enough to the operating profile, the estimation result will be proper for the final reliability estimation task because the hard-to-reach flowpaths during testing are also the flowpaths of less importance under real operating situation.

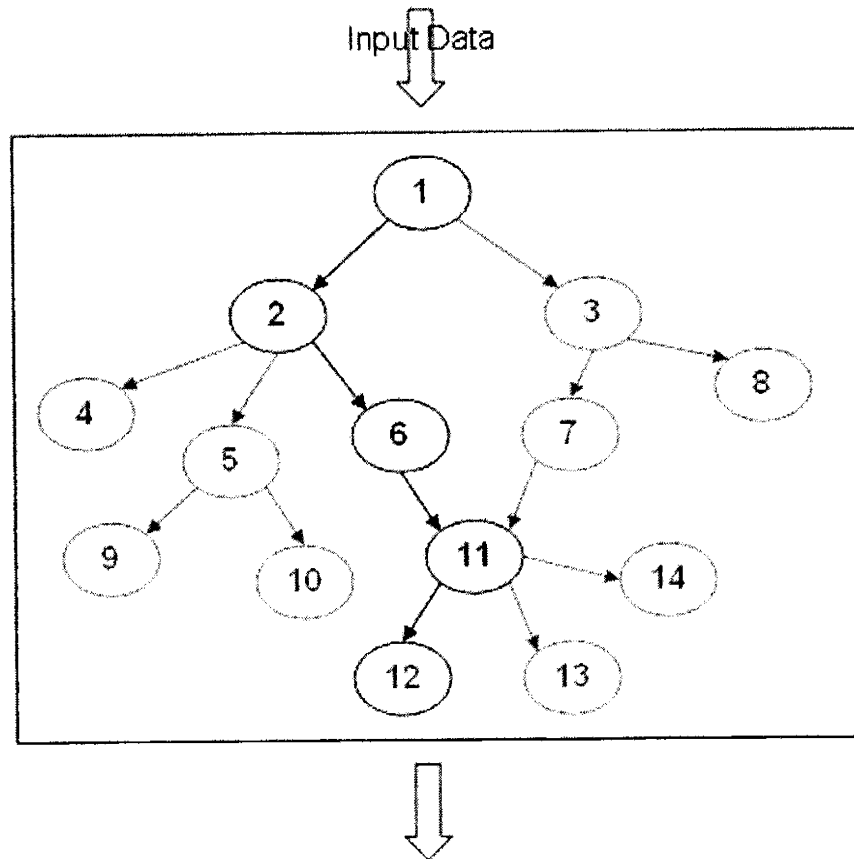


Figure 3-16 Identify Possible Flowpaths

We have seen the saturation effect in an earlier section when we discussed complete white box testing. According to the input data distribution, some flowpaths are visited 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 becomes more and more difficult to visit an untested flowpath. We call this phenomenon the “saturation effect” because although many new input data are applied, the speed of identifying new flowpaths is very slow. We can plot a diagram with the horizontal axis representing the

cumulative number of unique tests performed⁸, 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 becoming flatter and flatter. Imagine that at one point, we have tested all of the reachable flowpaths, the slope becomes equal to zero and stays zero after that point because whatever input data we apply after that point, no new flowpaths will be identified. 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 to extend the analytical curve to the saturation point, that is, to the point where the slope reaches a value of zero.

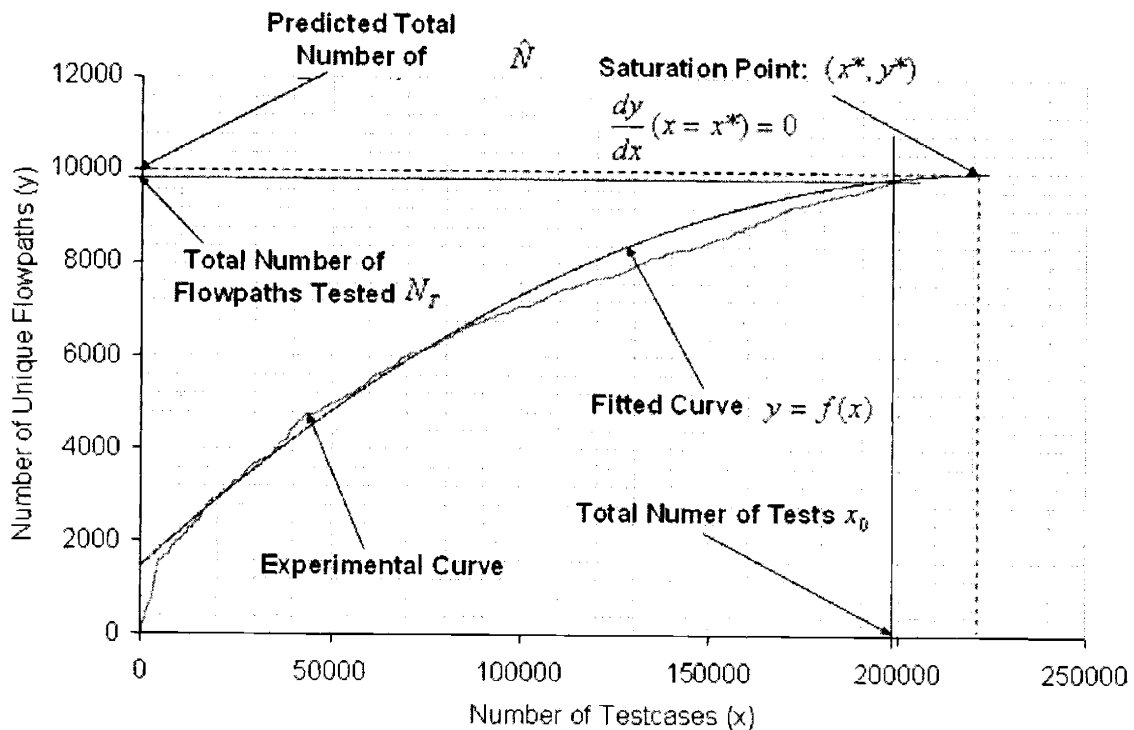


Figure 3-17 Estimate Total Number of Flowpaths, Experimental Approach1

In Figure 3-17, the program has been tested x_0 times⁹, out of which, N_T unique flowpaths are identified. The irregular curve is the experimental one. The smooth curve is the fitted curve obtained from the experimental one. Based upon the shape of the

⁸ 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.

⁹ x_0 tests are not necessary to be unique.

experimental curve, several functional forms can be used for the analytical fitted curve. A simple example would be a second order polynomial function, $y = f(x) = a + bx + cx^2$. The value of the parameters, a , b and c can be estimated from the experimental data. We call the point where the slope of the fitted curve becomes equal to zero the saturation point, after which, no new flowpaths will be identified because 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:

$$\frac{df}{dx}(x = x^*) = b + 2cx^* = 0 \quad \text{Eq. 3-91}$$

We obtain the result:

$$x^* = -\frac{b}{2c} \quad \text{Eq. 3-92}$$

The total number of flowpaths can be estimated as:

$$\hat{N} = y^* = f(x^*) = a + b\left(-\frac{b}{2c}\right) + c\left(-\frac{b}{2c}\right)^2 = a - \frac{b^2}{2c} + \frac{b^2}{4c} = a - \frac{b^2}{4c} \quad \text{Eq. 3-93}$$

Take the fitted curve as an approximation of the real experimental curve, the probability of visiting a tested flowpath next time can be estimated as:

$$p_{visit,T} = \frac{x_0}{x^*} \quad \text{Eq. 3-94}$$

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

Up to this point, we have described the first experimental method used to estimate values of the three important parameters, N , the total number of flowpaths that 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 upon its structure. It is a very simple approach with limited and simple calculations 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 because the difference between N and N_T is so small, that the noise of the fitted curve is much bigger than the difference. If this happens, this approach cannot be used to estimate the value of N , in which case, the second experimental approach has to be utilized. But the first approach is still useful in estimating the value of $p_{visit,T}$ and $p_{visit,U}$.

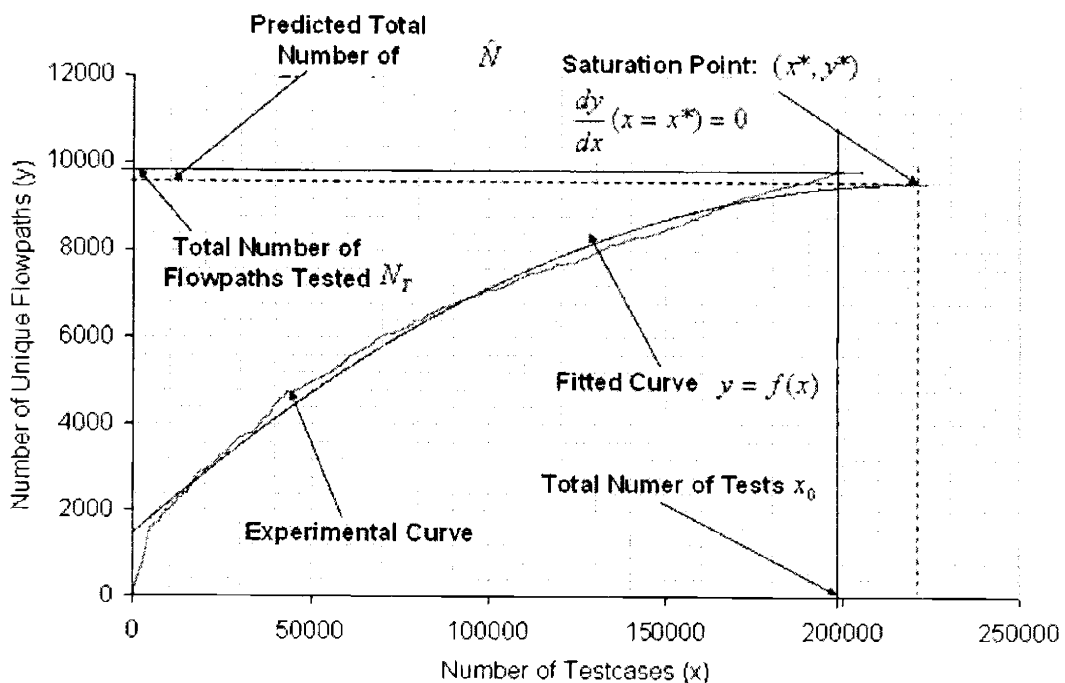


Figure 3-18 Predicted Total Flowpath Number \hat{N} Smaller than Tested Flowpath Number N_T

3.4.4.2.5.3. 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 the value of N , the total number of flowpaths, although, from the result, we can say that N is very close to N_T and we have almost tested all of the possible flowpaths. But neither “close” nor “almost” is a good or scientific answer. We wish to 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 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 as solid as the first approach.

Again, a necessary assumption used for the application of this method is that the input data are sampled randomly according to some distribution, in most cases, preferably, the operating profile. Some flowpaths are more easily visited than others. The ones that are easily reached will typically 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 will have more visits to them. We can sort the flowpaths according to the order in which they are identified. After many random tests, the number of visits to each flowpath can be plotted. We can imagine that there is a trend going through all the data points.

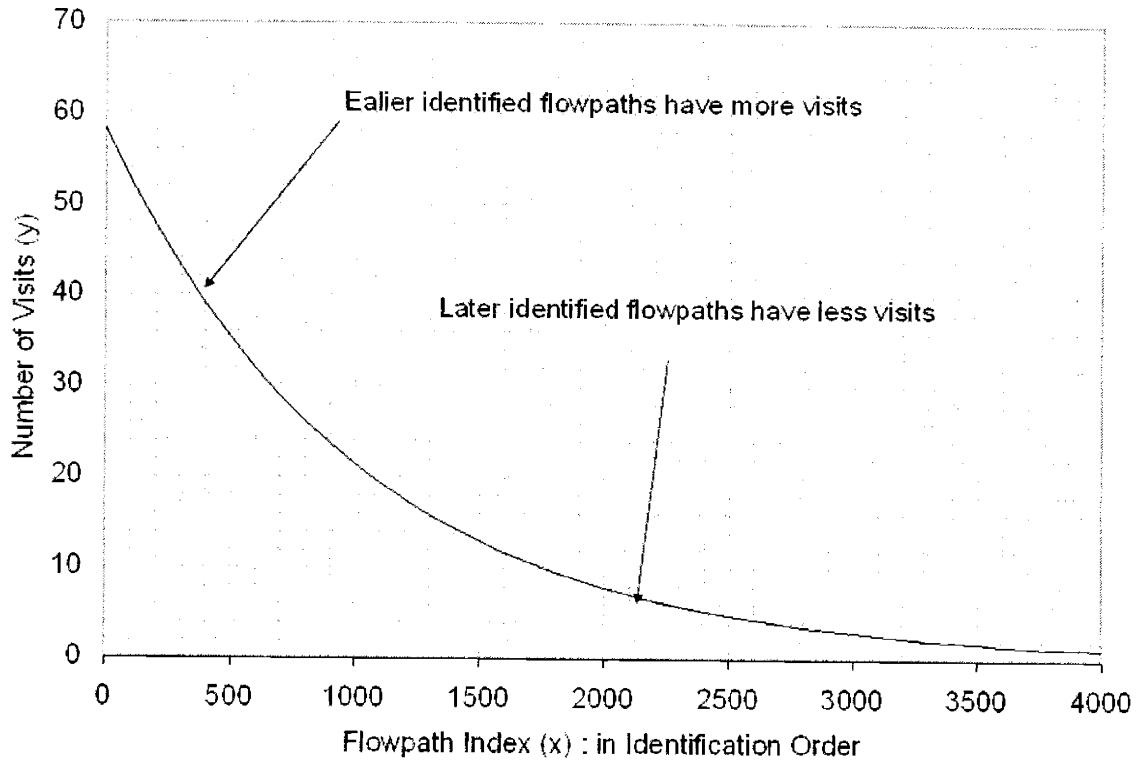


Figure 3-19 Illustration Ideal Number of Visits to Flowpaths According to Identification Order

In Figure 3-19, 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 to each flowpath. The flowpaths with smaller indices always have better chances of being visited during testing. If the input data follow the same distribution as the actual situation, the 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 indicates us the total number of flowpaths in the software. It is also possible that the curve does not intersect the horizontal axis, e.g. it has an exponential or asymptotic form. In this case, if we can connect this experimental curve with the visiting probability density distribution function of the flowpaths, some very useful results can be obtained.

For now, let us assume that we know the analytical form, $y = f(x)$, of the curve shown in Figure 3-19, 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 visits to each flowpath y is proportional to the probability of it being visited during a single execution of the program. The only difference between $f(x)$ and the probability density function $P(x)$ is a normalization constant. Thus, we have:

$$P(x) = \frac{f(x)}{\int_0^N f(x)dx} \approx \frac{f(x)}{\int_0^\infty f(x)dx} \quad \text{Eq. 3-95}$$

Where, N is the total number of executable flowpaths in the software.

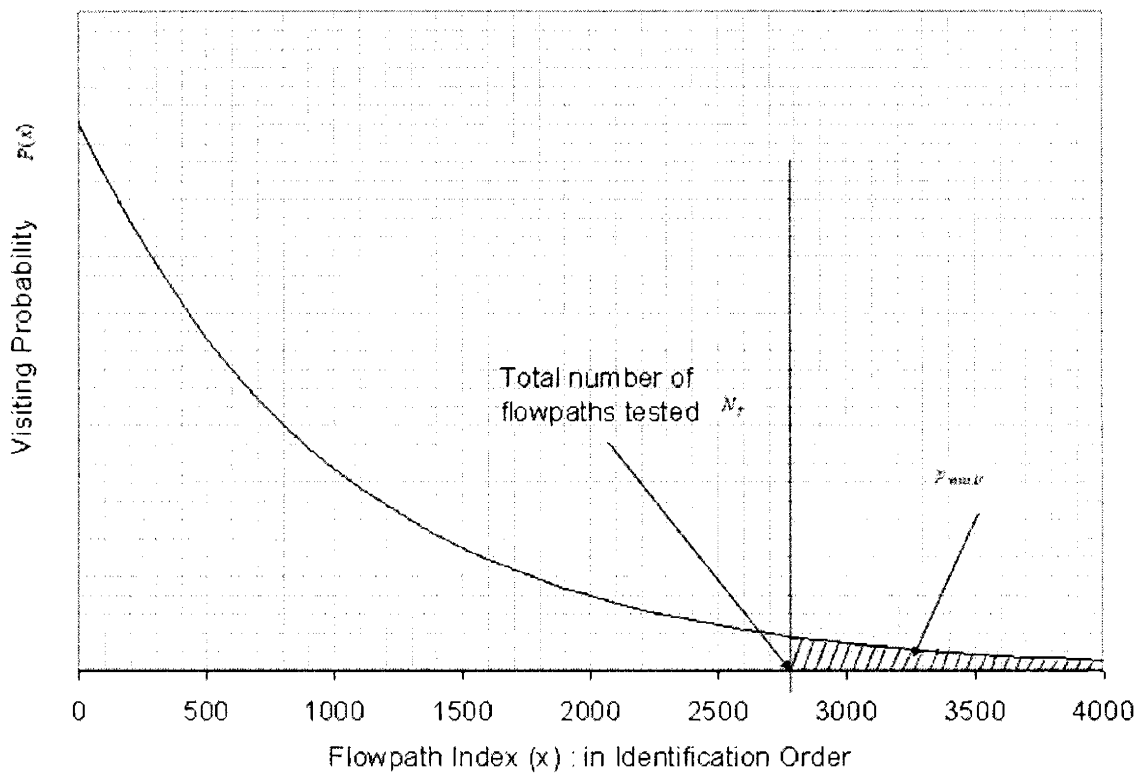


Figure 3-20 Illustration of Estimation of

the 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:

$$p_{visit,U} = \int_{N_T} P(x) dx \quad \text{Eq. 3-96}$$

All the discussions in this section until now are based on the assumption that we know such an ideal analytical function that connects the flowpath index and its number of visits during testing. But unfortunately, in reality, this is never true. We need to approximate such a function from our experimental data. In reality, although the trend is that earlier identified flowpaths tend to have more visits during testing, there are many variations among contiguous flowpaths regarding their number of visits. To put it another way, there exists considerable noise. 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, Eq. 3-96 can be used to make the estimation for $p_{visit,U}$.

There are two steps in going 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, then 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 because they are identified around the same order of discovery. For each group i , we calculate the average number of visits to each flowpath as:

$$\bar{y}_i = \begin{cases} \frac{1}{k} \sum_{j=1}^k y_j & i < n \\ \frac{1}{N_T - k \times (n-1)} \sum_{j=1}^{N_T - k \times (n-1)} y_j & i = n \end{cases} \quad \text{Eq. 3-97}$$

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:

$$\bar{x}_i = \frac{x_{i \times k+1} + x_{(i+1) \times k}}{2} \quad \text{Eq. 3-98}$$

The new plot has only $n \approx N_T/k$ data points with less noise. The proper value of k should be chosen according to the real experimental condition. If k is too big, the number of data points n will be too small and the resulting curve will not have enough statistical significance. If k is too small, then each group will contain only a very small number of data points to average and hence the noise-reduction effect will be very small.

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

$$y = ae^{-bx} \quad \text{Eq. 3-99}$$

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

$$\int_0^{\infty} ae^{-bx} dx = 1 \quad \text{Eq. 3-100}$$

$$\Rightarrow a = b$$

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

$$P_{\text{visit},U} = \int_{N_T}^{\infty} be^{-bx} dx = e^{-bN_T} \quad \text{Eq. 3-101}$$

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 Chapter3 is applied to a program, SVA, used in nuclear power plant's reactor coolant system. The generic Signal Validation Algorithm (SVA) is used to calculate and validate the values of parameters in the reactor coolant system. 001 CASE tool is adopted to facilitate and automate the methodologies. And it is not the only tool available for the desired purpose. The work described in this thesis is a continuation of work from an earlier Ph.D. thesis [Ouya95], where all the details about why 001 was chosen for this study were presented. The experiment reported here is aimed at investigating the feasibility of the proposed methods.

4.1.1. General

In this experiment, the original specification of SVA, written in plain English, is developed into C code using the 001 CASE tool following DBTF methodology described in Chapter2. After completion of building the models, variable and functional consistency as well as completeness is checked during the developments cycle. C code is automatically generated by 001 CASE tool. We call this code as sample code. The program then is tested against previously developed and tested software, obtained from our industrial collaborator, which is called oracle code. Test input data are sampled according to reasonable probability distributions obtained by examining the nature of the program. Should any discrepancy between the outputs from sample code and oracle code corresponding to the same input data occur, a failure of the sample code is assumed. The sample code is debugged. Retesting is performed for all the exploited input data in order to make sure that no new error has been introduced. During testing, the nodes and flowpaths are identified and recorded. Testing results and the identified flowpaths are

utilized to quantitatively estimate unreliability of the sample code, θ . Two Bayesian techniques are applied to make the estimation: the quick and approximate nodal coverage based method and the more precise refined feasible flowpath coverage based method. The detected errors and identified flowpaths are analyzed in the end.

4.1.2. Working Path Flow

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

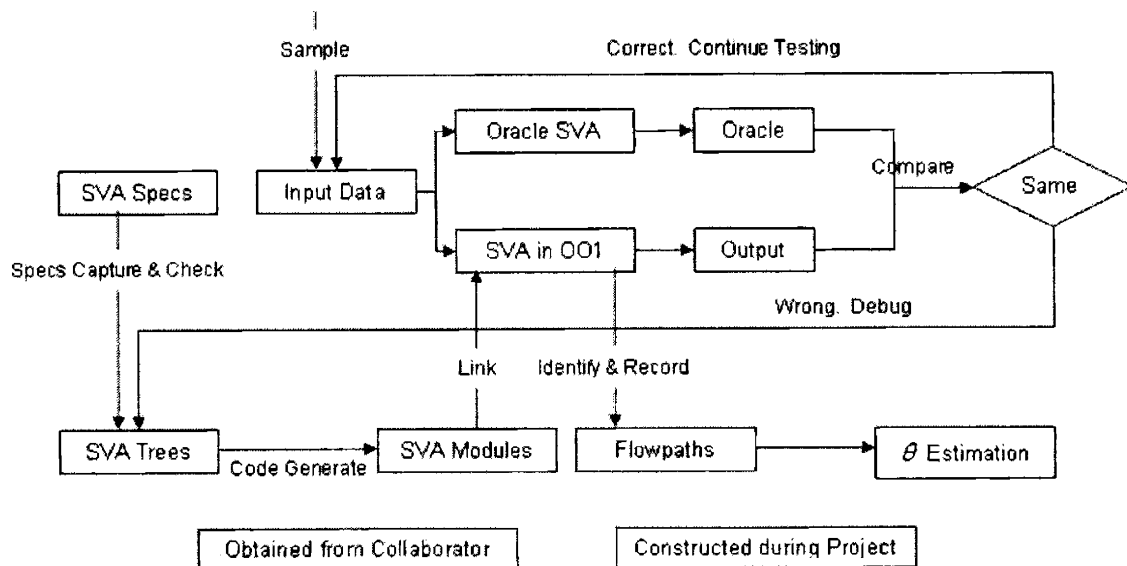


Figure 4-1 SVA Demonstration Flowpath

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 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 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 both normal and emergency operations. However, to simply display all these redundant measurements is not desirable because to do so may adversely complicate the operator's mental processes, especially data evaluation and analysis. One way to reduce the operators' workload in comparing, analyzing, and evaluating sensor signals is to apply automatic signal validation algorithms as a pre-filter. This reduces the amount of information passed to the operator.

The Signal Validation Algorithm (SVA) employed in this case study is a generic algorithm that performs signal validation for the plant processes that contain multiple sensors that measure 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 number of the sensors used.

4.2.1. SVA — the Algorithm

SVA is a generic validation algorithm that reduces unnecessary information. The algorithm takes the outputs of all sensors that measure 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 sensor readings and checks all sensor deviations against the average. If the deviation checks are satisfactory, the average is used as the "Process Representation" and the output is considered to be a "valid" signal. If any sensor readings do not successfully pass the deviation check, the reading with the greatest deviation from the average is deleted and the average is recalculated with the remaining

sensor readings. 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 use during emergency conditions, because 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 that is normally used for day-to-day work and with which the crew is most familiar with.

4.2.2. Original Design Document of SVA

There is an 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 that we used as a specification and from which the demonstration software is built with 001 CASE tool.

4.2.2.1. Oracle

Whenever a software testing task appears, there should be an independent mechanism, an oracle that is able to determine whether or not the results of test executions are correct. There is a large literature that addresses the question of how to generate an oracle automatically from both formal specification of the target software and tabulated values of the input variables. This is, by itself, a big topic in software engineering community. Oracle generation has not been a focus of the underlying works here. Thus, as in many software test and research practices, we assume that we have the

mechanism, an oracle that determines 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 from our industry collaborator. This is a “quick C” program of a little more than 3000 lines that is implemented onto IBM/PC. We term this piece of software “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 because it is developed under 001 CASE tool) is different from the output of oracle SVA, we claim a failure case. If this happens, debugging and retesting are performed on 001 SVA until all the outputs corresponding to all the tested input data are correct.

4.2.2.2. A Testing Document of SVA

The final document that 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 that is used as the oracle in the work we describe in this chapter. This evaluation was performed prior to algorithm implementation as the mockup to ensure that the validation algorithm in the mockup is based on a sound design.

The testing purpose, scope, background, algorithm definition, software implementation, etc. are covered briefly in the document. Ten test cases were designed and the desired outputs of the algorithm for the cases were identified prior to testing. Both the test cases and the correct outputs were based on operational requirements during both normal and emergency conditions that require the monitoring processes to contain multiple “like” parameters. Such redundancy is a prerequisite for signal validation. The snapshots of the testing screens, including both input and result screens were shown in the report. For the failures that occurred during testing, causes were analyzed.

According to the methods described in last chapter, this document is not essential for the purpose of demonstration. In this respect, the present document differs from the prior two documents, the software specification and the oracle program. Furthermore, the approach to select software input data in the report is not applicable to the approaches that we have described. As required by our thorough testing objective, automation of

every step is crucial to the success of the work reported here. But still, this document is very useful. This software evaluation document complements the software specification document in that it provides 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 provides some numbered examples about what 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. It is an automation of the Development Before The Fact (DBTF) approach. It is used to define and generate itself.

4.3.1. DBTF

As already discussed in Chapter 2, the Development Before The Fact (DBTF) approach is the formal software development technique that 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 that it is a preventative paradigm instead of a curative one. Problems associated with traditional methods of design and development are prevented "before the fact" 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 have surfaced at the most inopportune and expensive point in time.

4.3.2. 001 System

4.3.2.1. General

001 (pronounced "double oh one") is a fully integrated systems engineering and software design and development environment. It is the application of DBTF methodology. 001's motivation is to facilitate the "doing things right in the first place" development style and to avoid 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 a significantly lower error rate and significantly higher usability than traditional approaches.

4.3.2.2. 001 Features

4.3.2.2.1. System Oriented Objects (SOO)

All 001AXES models are defined and developed as System Oriented Objects (SOOs). A SOO is understood the same way without ambiguity by all other objects within its environment-including all users, models, and automated tools. Each SOO is made up of other SOOs. Every aspect of a SOO is integrated not the least of which is the integration of its object-oriented parts with both its function-oriented and its timing-oriented parts. The approach adheres to a philosophy which supports the theory that to integrate all the objects in a system you need be able to integrate all aspects of each object in the system.

4.3.2.2.2. Reusability

Every SOO is a candidate to be reusable and is inherently integratable within the same system, within other systems, and within all of these systems as they evolve. This is because every object is a system and every system is an object.

Capturing inherent and recursive reuse is provided by the formal definition mechanisms within the 001 AXES, as discussed later. Not only does 001 have properties in its definitions to find, create, and use commonality from the very beginning of the life cycle, it also has the ability to ensure commonality simply by using its language. This means that the modeler does not have to work at making something become object-oriented. Rather he or she models the objects and their relationships, as well as the functions and their relationships. The language inherently integrates these aspects as well as takes care of making those things that should become objects become objects.

Object oriented-like features such as polymorphism, encapsulation and inheritance-persistence formally reside both on the function side as well as the object side of a system where the functional side is defined in terms of the object side and vice versa. This provides the ability to trace within and between levels and layers of a system.

4.3.2.2.3. Traceability

001 provides traceability from requirements to design to implementation thereby rendering the system under development both manageable and maintainable. Systems being managed are objects from the viewpoint of 001. Objects in one phase, say, requirements, are traceable to objects in the next phase of development, the specification phase. This feature is helpful for large, complex software systems, for which maintenance in traditional environments is even more time-consuming and labor intensive than development.

4.3.3. 001 AXES Language

A formal systems specification language based on DBTF's foundations, 001AXES, is used to define a DBTF system. Based on a theory that extends traditional mathematics of systems with a unique concept of real-time distributed control, 001AXES has embodied within it a natural representation of structured relationships of objects and their interactions as events.

4.3.3.1. TMap and FMap

001 AXES language includes two main components, FMaps and TMaps. Every model in 001 is defined in terms of functional hierarchies (FMaps) to capture time characteristics and type hierarchies (TMaps) to capture space characteristics (Figure 4-2). A map is both a control hierarchy and a network of interacting objects. FMaps and TMaps guide the designer in thinking through concepts at all levels of system design. All model viewpoints can be obtained from FMaps and TMaps, including data flow, control flow, state transitions, data structure and dynamics. Maps of functions are integrated with maps of types.

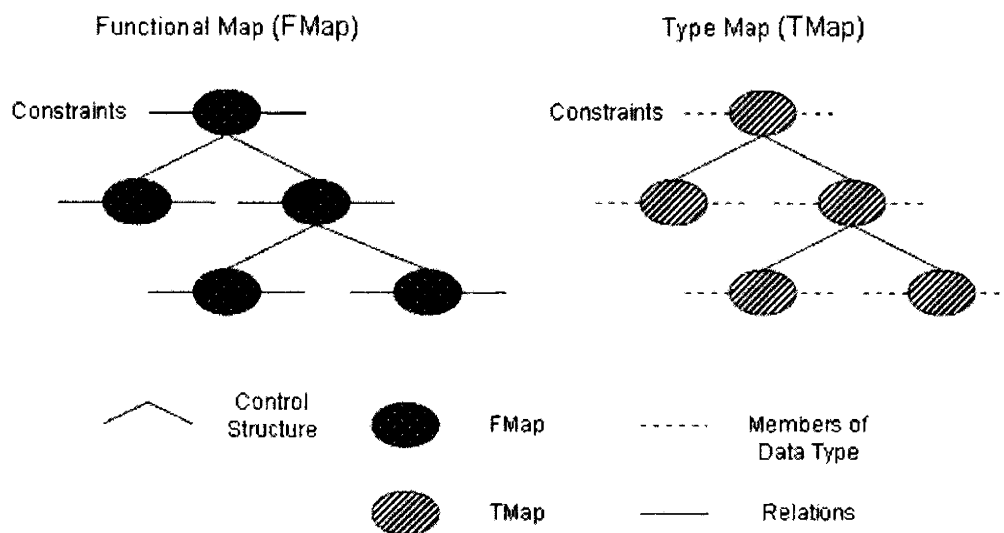


Figure 4-2 Functional Map and TMap in 001

On an FMap, there is a function at each node, which is defined in terms of and controls its child functions. For example, the function, “build the table”, could be decomposed into and control its child functions “make parts” and “assemble”. On a TMap there is a type at each node that is defined in terms of and controls its child types. For example, type “table” could be decomposed into and control its child types, “legs” and “top” (Figure 4-3).

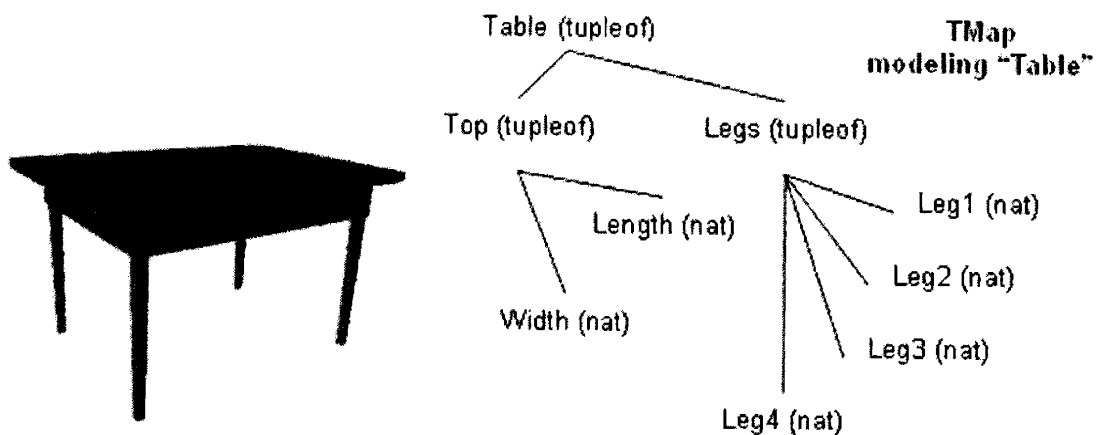


Figure 4-3 TMap Modeling the Concept of a “Table”

Every type on a TMap owns a set of inherited primitive operations. Each function on an FMap has one or more objects as its input and one or more objects as its output. Each object resides in an object hierarchy (OMap) and is a member of a type from a TMap. FMaps are inherently integrated with TMaps by using these objects and their primitive operations. FMaps are used to define, integrate, and control the transformations of objects from one state to another state (for example, a table with a broken leg to a table with a fixed leg). Primitive operations on types defined in the TMap reside at the bottom nodes of an FMap. Primitive types reside at the bottom nodes of a TMap.

When a system has its entire object values plugged in for a particular performance pass, it exists in the form of an execution hierarchy (EMap).

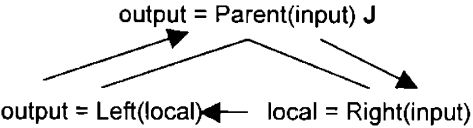
Once a TMap has been agreed upon, the FMaps begin almost to fall into place for the designers because of the natural partitioning of functionality (or groups of

functionality) provided to the designers by the TMap system. The TMap provides the structural criteria from which to evaluate the functional partitioning of the system. With FMaps and TMaps, a system and its viewpoints is divided into functionally natural components and groups of functional components which naturally work together. A system is defined from the very beginning to integrate inherently and make understandable its own real world definition.

4.3.3.2. Control Structures

All functions, or FMaps, that are not primitive are ultimately defined in terms of three primitive control structures: a parent controls its children to have a dependent relationship, an independent relationship, or a decision-making relationship. A formal set of rules is associated with each primitive structure. The behavior of the parent function is determined by the children functions that support it as well as by the control flow of information that the parent function imposes on its children.

The three primitive control structures are Join (J) for defining dependency between children, Include (I) for defining the independence of the children, and Or (O) for defining the children as alternatives that the parent function may invoke to perform its intended function. See Figure 4-4 for the control structures and their data flow rules.

Primitive Control Structures	Data Flow Rules
<p style="text-align: center;">JOIN</p> 	<p>The input variable lists of the Parent and Right child functions are identical including order;</p> <p>The output variable list of the Right and the input variable list of the Left functions are identical including order;</p> <p>The output list of the Left child and Parent functions are identical including the order.</p>

<p style="text-align: center;">INCLUDE</p>	<p>The rightmost part of the input variable list of the Parent and the input list of the Right child function are identical;</p> <p>The leftmost part of the input variable list of the Parent and the input list of the Left child function are identical;</p> <p>The children functions only receive their inputs from Parent's input variable list;</p> <p>The rightmost part of the output variable list of the Parent and the output list of the Right child function are identical;</p> <p>The leftmost part of the output variable list of the Parent and the output list of the Left child function are identical;</p> <p>Parent receives its outputs only from its children functions.</p>
<p style="text-align: center;">OR</p>	<p>The input variable list of the Parent, Partition, Right and Left functions are identical;</p> <p>The output variable list of the Parent, Partition, Right and Left functions are identical.</p>

Figure 4-4 Primitive Control Structures in 001

4.3.3.3. Parameterized Data Types

In 001, there are a number of pre-defined fundamental data types, e.g. int, nat, str, etc. More abstract data types are defined through parameterized data types. A parameterized data type is a defined structure that provides the mechanism to define a TMap without its particular relations being explicitly defined. Also, each parameterized data type assumes its own set of possible relations for its parent and child types. Each parameterized data type has a set of primitive operations associated with it for its use. Abstract data types which decompose with the same parameterized data type in a TMap inherit the same primitive operations.

There are three parameterized data types in 001. They are TupleOf data type, OneOf data type and OSetOf data type. The TupleOf data type is a collection of a fixed number of possible different types of objects. The OneOf data type is defined as a classification of objects of different data types from which only one data type is selected. The OSetOf data type is a collection of a number of variables of the same type of objects.

4.3.4. Software Life Cycle in 001

001 breaks the development life cycle into a sequence of stages, including: requirements and design modeling by formal specification and analysis; automatic code generation based on consistent and logically complete models; test and execution; and simulation. The process is described briefly in the following sections.

The first task is to define the model. Specification of the software is captured into FMaps and TMaps in either graphical or in textual form. Then the maps are submitted to the Structure Flow Calculator to automatically provide the structures and an analysis of the local data flow for a given model.

At any point during the definition of a model, it may be submitted to the Analyzer, which ensures that the rules for using the definition mechanisms are followed correctly.

When a model has been both decomposed to the level of objects that are designated as primitive and also successfully analyzed, it can be handed to the Resource Allocation Tool (RAT), which automatically generates source code from that model. RAT is generic in that it can be configured to interface with language, database, graphics, client server, legacy code, operating system, and machine environments of choice. The Type RAT generates object type templates for a particular application domain from a TMap(s).

The code generated by the Functional RAT is automatically connected to the code generated from the TMap as well as to code for the primitive types in the core library, and, if desired, libraries developed from other environments.

The generated code can be compiled and executed on the machine on which the tool suite resides; or, it can be ported to other machines for subsequent compilation and execution. User-tailored documents and metrics, with selectable portions of a system definition, implementation, description, and projections can also be configured to be automatically generated by the RAT. Once a system has been RATted, it is ready to be compiled, linked and executed.

The next step is to execute/test the system. One tool to use in this step is Datafacer, a run-time system that automatically generates a user interface based on the data description in the TMap. In the development of systems, Datafacer can be used in two major ways: as a general object viewer & editor and as a full end-user interface. The tool suite automatically generates a unit test harness incorporating Datafacer as a default test data set and data entry facility for the subsystem functions that are being developed. Datafacer chooses appropriate default visualizations for each data item that is to be displayed.

The Xecutor executes directly the FMaps and TMaps of a system by operating as a runtime executive, as an emulator, or as a simulation executive. As an executive, the Xecutor schedules and allocates resources to activate primitive operations. As an emulator of an operating system, the Xecutor dispatches dynamically bound executable functions at appropriate places in the specification. As a simulator, the Xecutor records and displays information.

4.4. Capture SVA Specification into 001 TMaps and FMaps

4.4.1. Formal Structures Building

In our experiment with SVA, we omitted the process of requirement definitions in the form of a user's English document. Rather, we used the same specification from out of which the oracle SVA program was built in order to compare outputs from both programs. Our first task was to transfer the specification of SVA into 001 TMaps and FMaps. Then FMaps and TMaps were 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 was performed, with special emphasis on key FMaps.

There are a total of 21 TMaps defined for SVA program. They can be subdivided 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 purposes, e.g. test case data structures, node and flowpath recorders, etc. Based on the 21 TMaps, FMaps are defined from SVA specifications. There are a total of 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 were found. Refer to Table 4-1 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 tested. Though both of them are testing procedures, they are quite different because of the size of the system under consideration. Because every module, or FMap in 001, is very small and simple, it is not very hard to perform a complete flowpath testing if it is necessary. However for the whole software, it is much more complicated and instead of a complete flowpath testing, some compromise is needed depending on the size of the program under question. In our experiment, we applied two very different strategies. In this section, the modular testing, the easier one is discussed briefly. The integrated testing task is left until the next section.

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 early stage.

For very important modules, feasible complete flowpaths testing can be performed. For a detailed description of complete feasible flowpath coverage testing, refer to [Sui98]. The process is summarized as follows. First, the flowgraph of the given module is constructed. In 001, it is merely 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 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 [Sui98], the whole process has to be performed manually. Because every module is relatively small and therefore has 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 sets of the module could then be created to test the underlying FMap. The correct output for each set of input data set is recognized manually. In this respect, the technique is the same as the complete feasible flowpath testing method.

Given the nature of the above two modular testing methods, the programmer who has created the module has to be the examiner of the module because refined knowledge about both the functions and the structures of the modules under testing is required. Though this is a contrary to a 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 very useful for 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 a fairly low failure rate. All the modules have been linked with 001 and are ready for an integrated test. Although modular testing is very important, the integrated test is indispensable. Because even if all the modules are performing perfectly according to their specifications, interface errors may still exist, which can never be detected until integrated testing is performed. It is almost never feasible to conduct 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 manually, which unfortunately turns out to be true in our case and even if all the constraints combine linearly, the undertaking is never trivial. In order to reduce the task, as already discussed in Chapter 3, we will reverse the step order. Instead of specifying a path and trying to find input values that cause the path to be traversed, we specify some input values and try to identify the traversed flowpaths. This can be done automatically. The overall effect of using this

approach is the same as identifying the paths first and choosing the input data later. This is what was described in Chapter 3 as grey box testing.

In the following subsections, the achievement of grey box testing on SVA is discussed in much more detail.

4.5.1. Test Data Generation

We have applied three different strategies for generating test data for integrated testing upon SVA 001 program. They are manual design of input data from operational profile / software specification, random sampling of input data based on functionality and software logic of SVA and manual identification of 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 thereby trying to cover most of the nodes; finally some input data again needs to be selected carefully to test the untested nodes. If we can discrete from the test results that, even though we have exercised a lot of tests on the program, the existence of untested flowpaths is un-ignorable. The more random sampling of the input data can be practiced to achieve a more complete result. Among the techniques to use, are two input data generation processes that are done by hand. 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 portions of the software (those nodes that have not been covered after quite a large amount of test cases are performed) would not be tested.

4.5.1.1. Input Identification from Operational Profile

Because we do not have any direct information about the operational profile of SVA, we have to refer to some auxiliary document. In this case, we choose the specification because most of the major logics are described clearly. The first step toward detection any failure involved with the most frequently occurring scenarios is to

design test cases to check these scenarios. All the input data in this category are selected manually. We developed about 300 test cases according to this technique. 2 failures happened. For more details about the encountered failures and detected errors, see the test results section 4.5.4.

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

4.5.1.2. Random Sampling

The technique that we use to generate the majority of the test cases we apply to SVA 001 program is described here. In total, we generated 198,321 test cases, out of which, 133,093 were 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 were 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 both narrow higher 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 that a validation fault or a 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 for the same input data set. These parameters all appeared, superficially at least, to be independent of each other. If this is the truth, our task would be much more straightforward. We would 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

by which we generate input data for the first narrow range group and then transfer it to the second wide range group by identifying the difference.

We start with the narrow range group by opting for a domain. This is done manually based on the specification. Because it seldom needs to be changed, the manual work is not a major effort 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 from the lower bound as:

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

Notice that we do not use *lb* and *ub* because 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.

A 40×4 array of independent numbers is randomly sampled between *lbR* and *ubR*, i.e.,

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

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 *kth* 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.,

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

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

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

Again, a 40×2 array of random numbers is generated for the forty sets of wide range sensor readings. Each set includes two sensor readings corresponding to the two PAMI sensors or wide range sensors.

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

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

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

Now, we have 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 readings, $s1, s2, s3, s4$, and two wide range sensor readings $p1, p2$; as well as four

criteria parameter values, iu, ev, iu_p, ev_p . The only inputs that 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 and narrow range. They are,

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

This means that 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 the wide and the narrow range are the same while the lower boundaries are different, i.e., $lb_p2 < lb2 < ub2 = ub_p2$. We also have,

$$\begin{aligned} lb_p3 &= lb3 = lbR, ub3 = ubR, ub_p3 = ubR_P \\ lb_p4 &= lb4 = lbR, ub4 = ub_p4 = ubR \end{aligned} \quad \text{Eq. 4-10}$$

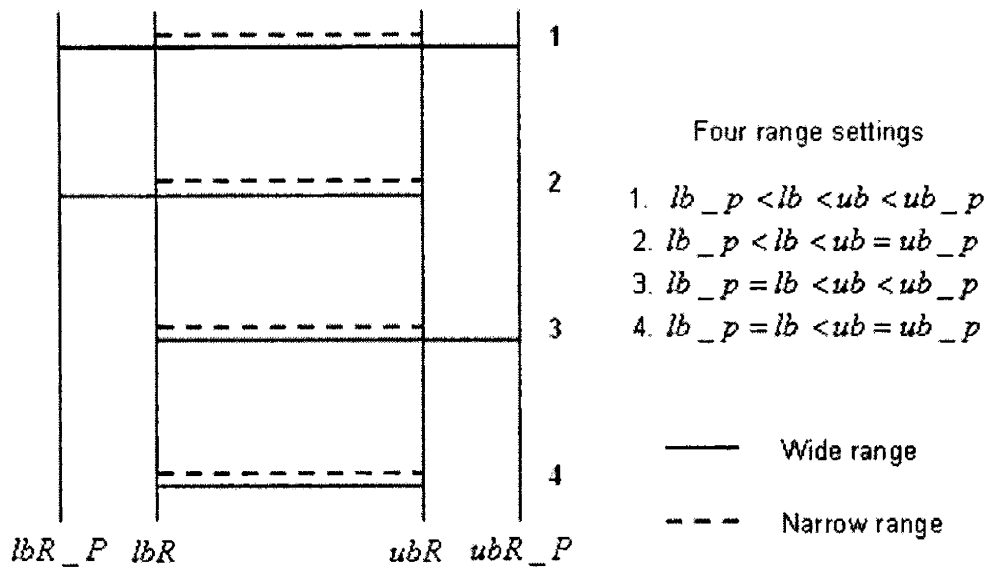


Figure 4-5 Four arrangement settings for one SVA input data set

Figure 4-5 shows the four arrangement settings used during SVA testing. Every one of the above generated forty ten-random-number input data sets was put into test against the four arrangement settings described.

Do we have all the required input data for testing? If we were to do interactive testing, then we require one person to behave like an operator. When the SVA program prompts the operator to choose a representative sensor reading, this person responds according to the instructions. In order to save time, as we will mention in the next section, we have excluded all the human-computer communication. Except a few cases where 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. This action simulates human behavior. Because 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 the number zero is chosen, it means that 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 selected sensor (1). Out of the above described input data generating process, we can get 40×4 input data sets.

In generating test cases for the SVA 001 program, besides the above random sampling method, we have also used another approach, which is only a little different. We employ a Gaussian distribution instead of a uniform distribution for all the parameter values obtained through the random sampling process. In reality, the Gaussian distribution method was the first method that we adopted in our test data generation process. After some tests, we concluded that a more diversified input data was more effective in testing untested flowpaths and hence changed all the underlying distributions from Gaussian to Uniform. See Table 4-2 for the parameter random sample methods that we used 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	$pi = uniform(lbR_P, ubR_P)$	$pi = Gaussian(cen_p, var_p)$
iu_p, ev_p	$iu_p, ev_p = uniform(CRI_MIN_P, CRI_MAX_P)$	GIVEN
op	$op = uniform(0,6)$	$op = uniform(0,6)$

Table 4-2 Random Sampling Input Data for SVA

The above random input data generation process permits a huge number of input data sets to be created in very little time.

4.5.2. Automatic Testing Process

We want to test our program as thoroughly as possible because of 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 to the oracle and target software, automated oracle output and test 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

This section describes how to achieve automatic input data reading for both the oracle generating program and the sample software program.

4.5.2.1.1. Oracle Input Reading

The original SVA program in quick C that is 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 testing screen. The original interface can be seen in Appendix 7.3. This I/O process is extremely time-consuming, and this impedes large number of tests. In order to develop a thorough but practical testing regimen for the sample software, SVA, we modified the original SVA software to omit the portions concerned with input and output readings from the screen, thereby concentrating efforts 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 7.4.

4.5.2.1.2. 001 SVA program Input Reading

In SVA, the evaluations are conducted continuously. That is, if no terminating command is received, the next set of input data is evaluated using the values of all the parameters obtained from the last evaluation process. In order to ease flowpath record keeping for each test case, we purposely broke the continuous process in the 001 SVA program into single processes. But because of the continuity in the oracle SVA program and because we have to compare results from both programs for correctness checking, we have to make sure that both programs are under the same initial condition 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 during execution of the oracle SVA program. This was achieved is that during execution of the oracle SVA program by generating a new input data file, including both the actual input data and the intermediate parameter values that is ready to be converted to 001 SVA readable input data file. Refer to Appendix 7.5 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 7.6. The resulting input files for the 001 SVA program are named as “#.#.complete_in.omap”.

In the 001 Tool introduction section, we 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 that is provided automatically by the 001 Tool Suite takes much more time than could be afforded. After about one thousand test cases, it becomes a bottleneck. So this graphic interface was removed. Some work is done within 001 Tool in order to avoid the interface generating step, which leads to the 001 SVA program’s direct reading of its input data files.

4.5.2.2. Output Recording

In order to automate the checking of test results, the results from both oracle and 001 program are documented as output files instead of using a screen to display the results. This is done by keeping the formats of both types of output files uniform, making it very easy to compare test results from 001 program with the corresponding oracle.

4.5.2.2.1. Oracle Output Recording

The original oracle SVA program displays results on screen as described in Appendix 7.3. With the aim of automatic checking of the test results, 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 standards to check if those output files from the 001 SVA program are correct. Refer to Appendix 7.7 for the format of the “oracle#.dat” files.

4.5.2.2.2. 001 SVA program Output Recording

The 001 SVA program reads the input data sets directly from “#.complete_in.omap” files. The resulting outputs are directed into files named “output#.#”, which have exactly the same format as “oracle#.dat” presented in Appendix 7.7.

There is yet another problem in comparing “oracle#.dat” and “ouput#.#” even though they are of the same format. For the oracle SVA program, the algorithm runs 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 from one input file recorded in a single output file “oracle#.dat”. But for the 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 with a program written in PERL, “output2o.pl”. The merged output files from the 001 SVA program are named as “o#.dat”. The “o#.dat” files from the 001 SVA program can be compared directly to the “o#.dat” files from oracle SVA program.

4.5.2.3. 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 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, the message “There are totally 0 differences.” in “result#.dat” will be displayed. 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 that all the detected errors are removed immediately without introducing new defects. In this software testing and reliability estimation study, because of the nature of the software under consideration, perfect error removal becomes a requirement instead of an assumption. It is irresponsible to say that we assume that we have eliminated the encountered error without introducing any new error. We have to prove that this is true. The technique we apply here is a very simple and a traditional one. Namely, 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. 12 errors were detected throughout the entire testing process. All of the errors were corrected immediately and regression testing was performed. In other words, 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 details of all the encountered errors, refer to Appendix 7.8.

4.6. Reliability Estimation

4.6.1. Required Information

It is very important to make good approximations about the underlying software reliability after completing some tests on it. Otherwise, we will not 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 if further action is wanted. Before we make the reliability estimation for 001 SVA program, let us make a summary about the information that is needed.

For the nodal coverage based reliability estimation approach, we need:

- A list of unique nodes on the SVA program¹⁰;
- Visiting frequency of each node during testing;

For the 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 are provided about how we get the above information from the 001 SVA testing processes and use it to estimate the reliability of 001 SVA program. Though, in this thesis, only the final reliability estimation results are presented, the same reliability techniques can be applied to the target software at any

¹⁰ It is required, in the nodal coverage reliability estimation approach, that all the nodes are tested at least once.

point during the 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 previously defined, 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 sufficient to identify the tested flowpaths is retained.

4.6.2.1. The Nodes in 001 (two types: functional/decision-alternative)

In 001, all the functions are defined as FMaps. For this reason, we use nodes instead of statements in this section. All the nodes within 001 can be divided into two types: decision nodes and functional nodes. To use 001 language, all the decision nodes should have at least two child nodes. The fundamental control structure between a parent decision node and its child nodes are OR (O). After executing a decision node, one and only one of its child nodes (called alternative nodes of their parents) will be executed. A path is uniquely recorded only by remembering all the decision nodes and their chosen alternative nodes along that path. For all the other nodes, because there is no decision to be made, 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 designate 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 to record a special group of them. Except decision nodes and their chosen alternatives, we also record the first node of every FMap 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 know to which FMap they belong. This is the same as whenever a decision node is recorded. That is, besides its name, its residing subroutine name is also recorded. Thus, the information we get for one flowpath is:

```
FMap head1
[decision node 1 → alternative1
 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

In order to ensure complete path coverage, it is necessary to expand the program flowgraph so that every distinct execution path is tested. In the event of looping structures, this expansion of the flowgraph structure may be unfeasible because of the complexity of the graph. In such a case, only limited testing may be practical. For this reason, we formulated a feasible version of flowpath coverage testing, which is limited to two passes through each loop. We have found that this approach will reveal many of the errors in a program. A program named “cut_iterations” using 001 CASE Tool is built to

serve this purpose. It cuts 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.

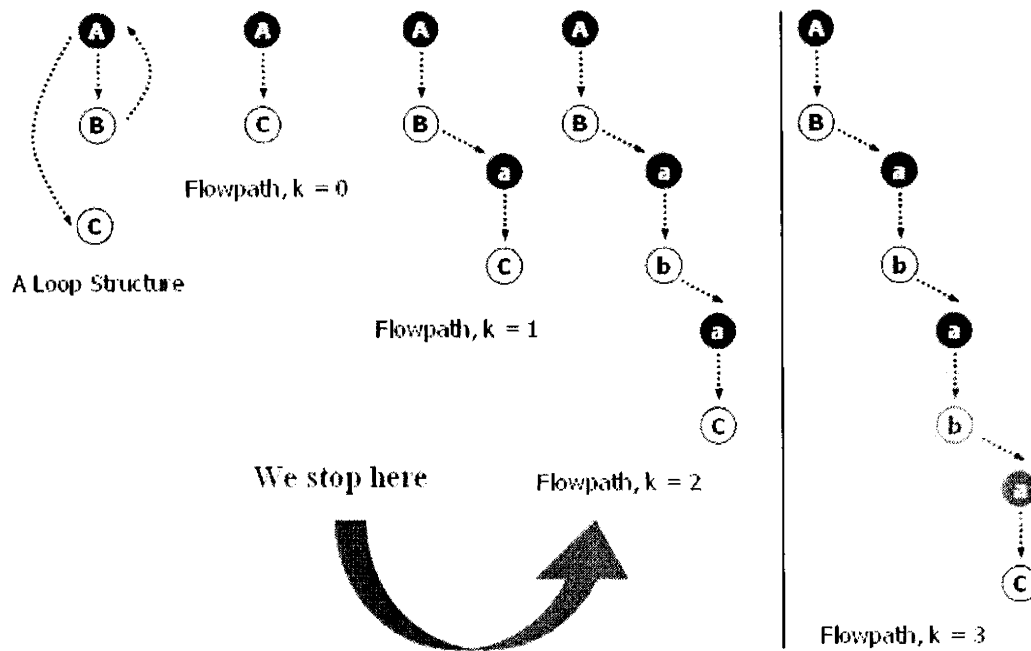


Figure 4-6 Cut Extra Iterations of a Looping Structure

4.6.3. Reliability Estimation Process Overview

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

4.6.4. Complete Nodal Coverage Approach

In this approach, it is required that all the nodes¹¹ be 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 in 001 to estimate the visiting frequency of every node within the 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 has a zero visiting frequency during the passing tests, special investigations are performed. Eventually all the existing nodes that do not belong to the 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

A tool is developed under 001 to find out both the nodes that have not been covered and the visiting frequency of the already tested nodes. The name of the tool is “search_uncovered_simple”¹². It takes in both the static un-expanded 001 FMaps of the target software and the recorded flowpaths of every test case as inputs. The methodology of the tool program is described in Chapter3. The output of the program is a list of the nodes of the target software with the number of visits and visiting frequency attached after the name of the nodes.

We used the tool program “search_uncovered_simple” after 78,717 test cases¹³. Before that, we kept track of the speed of increase of new detected flowpaths. 78,717 itself is a big number, nevertheless unique flowpaths were still showing up at a steady rate. So we decided to check if all the nodes had been covered and if the newly detected flowpaths were only new combinations of already tested nodes. As result of the node visiting frequency calculation, we found 16 unvisited nodes after the 78,717 test cases. Because 16 is not a big number, we managed to investigate each of them. They were divided into two groups after the investigation. The nodes in the first group are the nodes

¹¹ If the software is developed under 001 Tool Suite environment, it is not required that all the 001 built-in library nodes be 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.

¹² For details about “search_uncovered_simple”, refer to Appendix7.9.

¹³ Test case number 0.1 to test cases number 1999.40.

that can only be reached by very special 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 by randomly sampled data. The logic behind each node in this group is studied carefully and purposely designed input data was applied to 001 SVA to test these nodes. We found that 14 out of the 16 unvisited nodes belong to this group. There were 2 uncovered nodes left. They were extra nodes which we decided can never be reached by any input data and thus do nothing of benefit to the program. They were deleted from the 001 SVA program. For details about all 16 unvisited nodes that were found after 78,717 tests on the 001 SVA program, refer to Appendix 7.10. We found 2 errors from the 14 nodes in the first uncovered node groups tested by the 84 test cases. Compare with 10 errors that were found from the previous 78,717 tests, the error detection efficiency is very high. Thus, this is a very effective method to identify errors. For details of the errors, see the next section, where the software reliability is estimated based on flowpath coverage. The error information is not needed to estimate software reliability because any detected error should be fixed immediately under the safety-critical requirement.

In 001 SVA program, there are 432 SVA nodes in total. Look at Figure 4-7 for number of visits to SVA nodes.

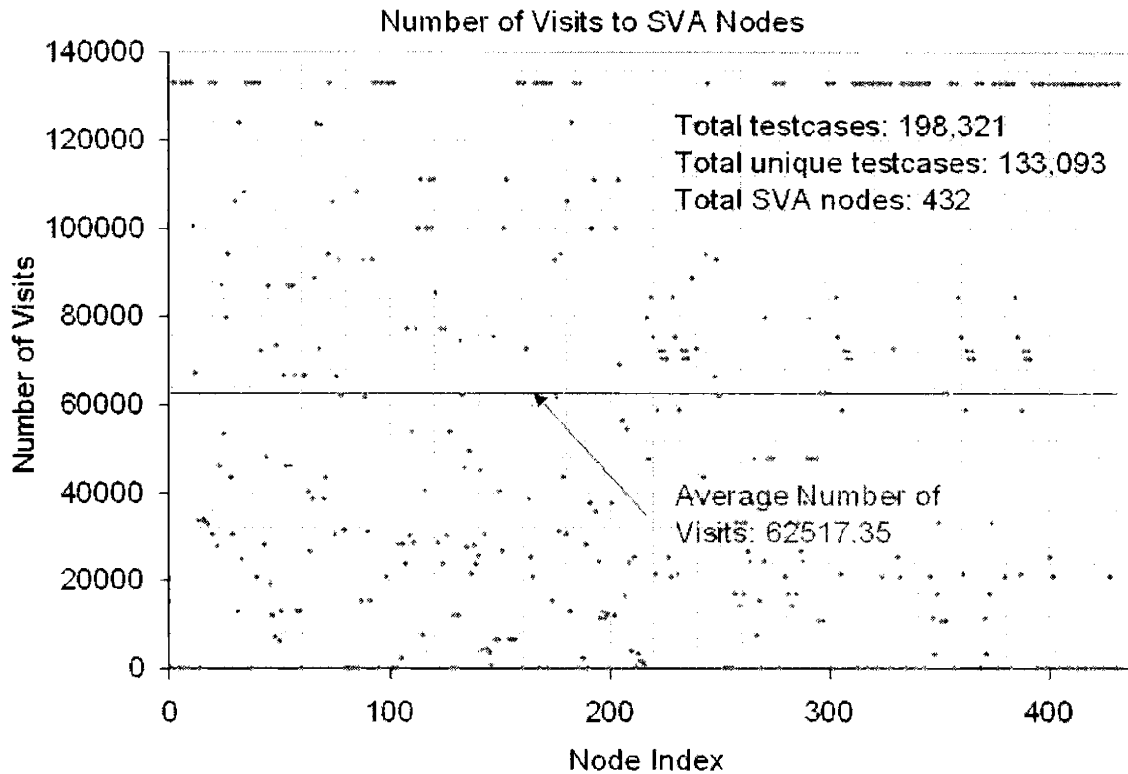


Figure 4-7 Number of Visits to SVA Nodes after 198,321 Tests

4.6.4.2. Testing Results, from Nodal Coverage Point of View

Overall, we tested the 001 SVA program 198,321 times, out of which 133,093 tests were unique. There were 1063 nodes, among which 432 nodes were created by the programmer. In total, the 1063 nodes have been tested 38,013,756 times by unique tests (133,093); 27,007,494 tests hit the 432 SVA nodes and $(38,013,756 - 27,007,494) = 11,006,262$ tests hit SVA built-in nodes. The average number of visits to SVA nodes is $(27,007,494/432) = 62,517.35$.

12 errors were found during the whole testing process. 2 of them, as described in the last section, were detected by the input data designed to cover the untested nodes (testcase2000.1 ~ testcase2010.20, 84 test cases, inputs generated manually); 10 of them were identified before the process (testcase0.0 ~ testcase1999.40, 78,717 test cases, inputs generated manually and automatically); No error was 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. The same reliability estimation method can be performed at any moment during the testing process. If there have been any modifications since the last reliability estimation, the reliability estimation is a recalculation because the underlying reliability level has been increased because of the perfect removal of errors. If no new error has been 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 because of the additional tests performed on it.

According to the derivation given in Chapter3, the unreliability probability distribution of node i_n , which has been tested s times without any failure, based on a uniform prior unreliability distribution for the node, is given by:

$$f(\theta'_{i_n}) = \frac{(1 - \theta'_{i_n})}{B(1, 1 + s)} \quad \text{Eq. 4-11}$$

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

$$\bar{\theta}'_{i_n} = \int_0^1 f(\theta'_{i_n}) d\theta'_{i_n} = \frac{1}{2 + s} \quad \text{Eq. 4-12}$$

¹⁴ This probability is obtained through Bayesian updating method with uniform prior probability distribution for θ'_{i_n} , i.e. $f_0(\theta'_{i_n}) = 1$. 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} .

where s is the number of tests applied on node i_n . For each of the unique SVA nodes, we calculated its unreliability in accordance with Eq. 4-12. The unreliability values are shown in the following figure.

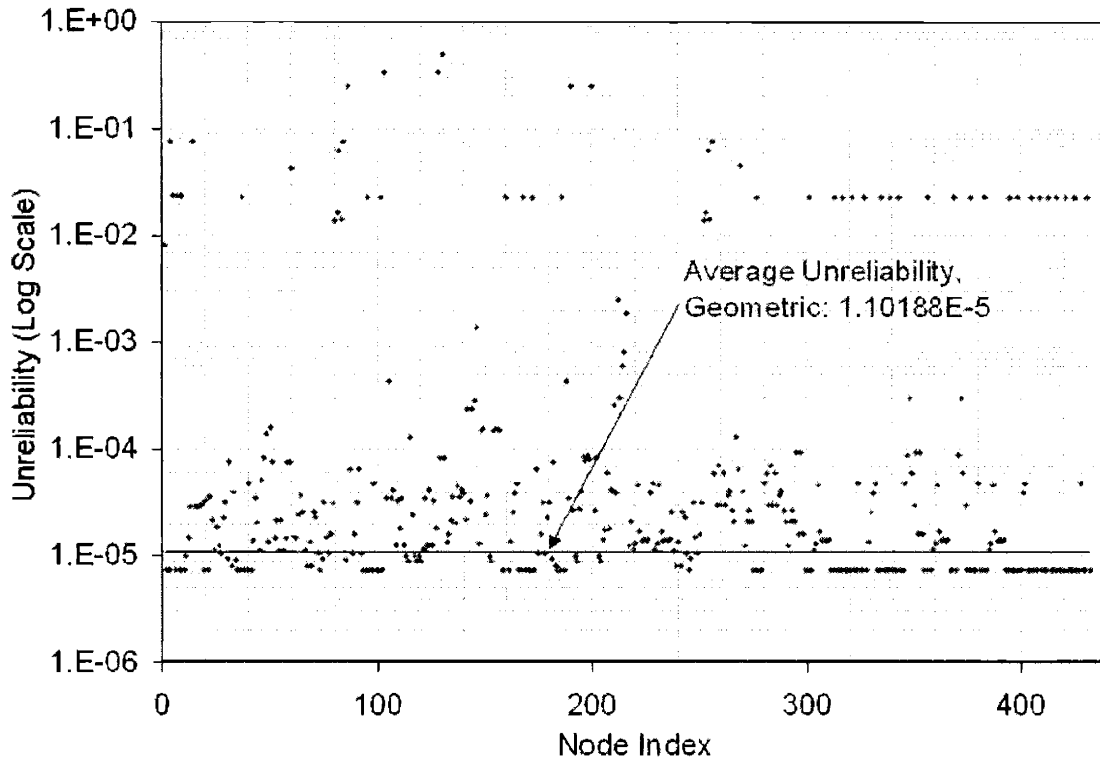


Figure 4-8 SVA Unique Node Unreliability

As seen in Figure 4-8, the geometric average unreliability of all the SVA nodes is $1.10188E-05$. The average number of SVA nodes visited per execution in SVA is the total number of visits to all the SVA nodes, i.e., 27,007,494, divided by the number of total tests performed, i.e., 133,093. Thus,

$$\bar{x}_{SVA} N_{SVA} = \frac{27,007,494}{133,093} = 202.93 \quad \text{Eq. 4-13}$$

The overall unreliability of the SVA nodes is:

$$\theta_{n(SVA)} = 202.93 \times 1.101882 \times 10^{-6} = 2.23596 \times 10^{-3} \quad \text{Eq. 4-14}$$

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

$$\theta_n = \frac{27,007,494}{38,013,756} \times \theta_{n(SVA)} + \left(1 - \frac{27,007,494}{38,013,756}\right) \times \theta_{n(001)} = 1.5886 \times 10^{-3} \quad \text{Eq. 4-15}$$

According to nodal coverage reliability estimation method, the unreliability of 001 SVA program is:

$$R_n = 1 - \theta_n = 1 - 1.5886 \times 10^{-3} = 0.9984 \quad \text{Eq. 4-16}$$

So far, we have presented the process and result of unreliability estimation for the SVA program. It is very straightforward to understand and simple to perform. But, we still have not found a good technique to incorporate any prior distribution for any of the nodes unreliability within the program. Thus, the result shown above is not comparable with the result obtained from the flowpath coverage based reliability estimation method.

Despite the weakness of the 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 to put more energy into the more significant nodes. Furthermore, as can be seen from the error data, searching for the input data to cover those nodes that have not been tested after a lot of testing makes it easier to capture unexpected errors in the software. Also surplus nodes that may serve no purpose in the software can be spotted during the process.

In the end, we want to present a quick way to estimate software unreliability incorporating nodal coverage testing information and the average prior unreliability value.

We do not have enough information to formulate a prior density distribution for the unreliability of a single SVA node in this study. The only prior testing piece of information we know about SVA program is that similar software system has an average unreliability value of 0.01. As discussed in the flowpath coverage based reliability estimation approach, we can directly use this prior average value to perform Bayesian updating process. But what if we only have the average value and the nodal coverage information about a software program and we want to use these only two pieces of information to get a better approximation than what we can get by assuming a uniform prior distribution for software unreliability?

Recall the formula to estimate the unreliability value of a flowpath with a prior average value of an error existing probability Pe_0 . If the flowpath has been tested s times failure free, its unreliability can be updated as:

$$\bar{\theta} = \frac{p_{f,n}(1-p_{f,n})^s Pe_0}{p_{f,n}(1-p_{f,n})^s Pe_0 + (1-Pe_0)} p_{f,n} \quad \text{Eq. 4-17}$$

Now let us look at the software as made up only by one flowpath (with its unreliability value the average of all the flowpaths). First, if we update the unreliability value of this one flowpath software based on a uniform prior distribution, $\bar{\theta}_{p,0.5}$, with average value of 0.5, we should obtain about the same unreliability value as we can get from the nodal coverage based approach based on a uniform prior, $\bar{\theta}_{n,0.5}$. This can be written as:

$$\bar{\theta}_{p,0.5} = \frac{p_{f,n}(1-p_{f,n})^s \times 0.5}{p_{f,n}(1-p_{f,n})^s \times 0.5 + (1-0.5)} p_{f,n} = \frac{p_{f,n}(1-p_{f,n})^s p_{f,n}}{p_{f,n}(1-p_{f,n})^s + 1} \cong \bar{\theta}_{n,0.5} \quad \text{Eq. 4-18}$$

$\bar{\theta}_{n,0.5}$ in our SVA example is equal to 1.5886E-3. From Eq. 4-18, some flowpath information can be backed out. We use x to simplify Eq. 4-18:

$$x = p_{f,II} (1 - p_{f,II})^s \quad \text{Eq. 4-19}$$

$$\bar{\theta}_{p,0.5} = \frac{x p_{f,II}}{x + 1} \cong \bar{\theta}_{n,0.5} \quad \text{Eq. 4-20}$$

Use the condition that $x \ll 1$ and plug into Eq. 4-20, we get:

$$\bar{\theta}_{p,0.5} = x p_{f,II} \cong \bar{\theta}_{n,0.5} \quad \text{Eq. 4-21}$$

Now let look at how the reliability estimation result is going to change if we use a prior average that is much less than 0.5. Substitute Eq. 4-19 into Eq. 4-17, we get:

$$\bar{\theta}_{p,Pe_0} = \frac{x P e_0}{x P e_0 + (1 - P e_0)} p_{f,II} \quad \text{Eq. 4-22}$$

Use $P e_0 \ll 1$ and $x \ll 1$, we get:

$$\bar{\theta}_{p,Pe_0} \cong \frac{x}{x + (1 - P e_0) / P e_0} p_{f,II} = \frac{x}{0 + 1 / P e_0} p_{f,II} = x p_{f,II} \times P e_0 \quad \text{Eq. 4-23}$$

Compare Eq. 4-21 and Eq. 4-23, we get:

$$\bar{\theta}_{p,Pe_0} \cong Pe_0 \bar{\theta}_{p,0.5} \cong Pe_0 \bar{\theta}_{n,0.5} \quad \text{Eq. 4-24}$$

From Eq. 4-24, if we only know the average prior unreliability value and the nodal coverage information, we can use the product of the average prior and the average unreliability value obtained from nodal coverage based Bayesian updating process to get an approximation. In the SVA example, the posterior average unreliability estimated via. Such approximation is:

$$\bar{\theta}_{p,Pe_0} \cong Pe_0 \bar{\theta}_{n,0.5} = 0.01 \times 1.5886 \times 10^{-3} = 1.5886 \times 10^{-5} \quad \text{Eq. 4-25}$$

The corresponding reliability value is:

$$R_{p,Pe_0=0.01} = 1 - \bar{\theta}_{p,Pe_0} = 1 - 1.5886 \times 10^{-5} = 99.99841\% \quad \text{Eq. 4-26}$$

This result is comparable with the result estimated from the flowpath coverage based unreliability estimation result in next section.

4.6.5. Refined Complete Feasible Path Testing Approach

Now, consider the second reliability estimation method, the refined complete feasible flowpath based reliability estimation method. This is a finer approach, which gives a better estimation, but also requires more information from the software under consideration. For safety-critical software, which is often relatively simple¹⁵, this method is recommended compared to the nodal coverage based approach.

¹⁵ For “relatively simple software”, it should be possible to test a significant percentage of all feasible flowpaths.

4.6.5.1. Truncate Extra Iterations of Looping Structures

With the aim of making the complete flowpath testing method 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. Because we now adopt the opposite way to achieve the flowpath coverage objective, i.e., we randomly sample the input data first and identify the triggered flowpaths during execution, we have to accept whatever flowpaths are recorded during the testing process. Instead of cutting the 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. Any flowpath that contains more than two iterations of a loop is treated as though it has only two iterations of the loop.

With the intention to cut extra iterations of every looping structure for every recorded flowpath, a tool program named “cut_iterations” under 001 was 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 flowpaths 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 of the extra iterations cut. After this step, unique flowpaths will be identified from these new cut flowpath records. For details about the “cut_iterations” tool, refer to Appendix7.11.

4.6.5.2. Distinct Feasible Flowpath Identification

Once the surplus repetitions have been cut, it is time to identify the unique flowpaths that have been tested. Originally, a 001 tool was built to perform this function. 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 and compare every

corresponding node from both flowpaths. 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. Because every flowpath requires about 40Kb and because of the overhead of 001 for its safety features, the process turns out to be both time and memory intensive. The process of comparing only 1597 flowpath files causes the available memory to be exceeded. In order to solve this problem, two changes are made.

First, instead 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. Then, some of the groups are combined, after which, we have a fewer 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. All 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. The Linux shell commands and several programs written in PERL are used for the file comparison task. Refer to Appendix7.12 for details.

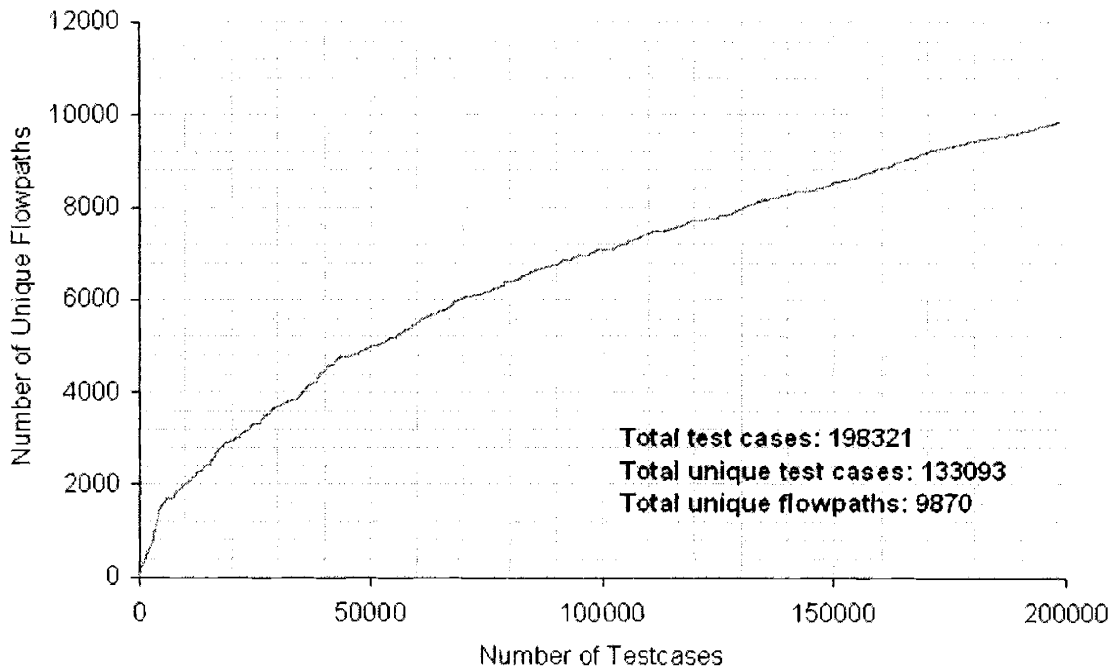


Figure 4-9 Number of Test Cases vs. Number of Flowpaths, SVA

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

4.6.5.3. Count Visiting Frequency of Flowpaths

To estimate the unreliability of every tested flowpath, we need to know the group of unique flowpaths and how many times each flowpath has been tested. Again, in order to make better usage of the system resources, we turn to PERL instead of 001. Also, because of the number of flowpaths we were dealing 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 7.13. The following figure shows the visiting frequency of all the tested 001 SVA flowpaths.

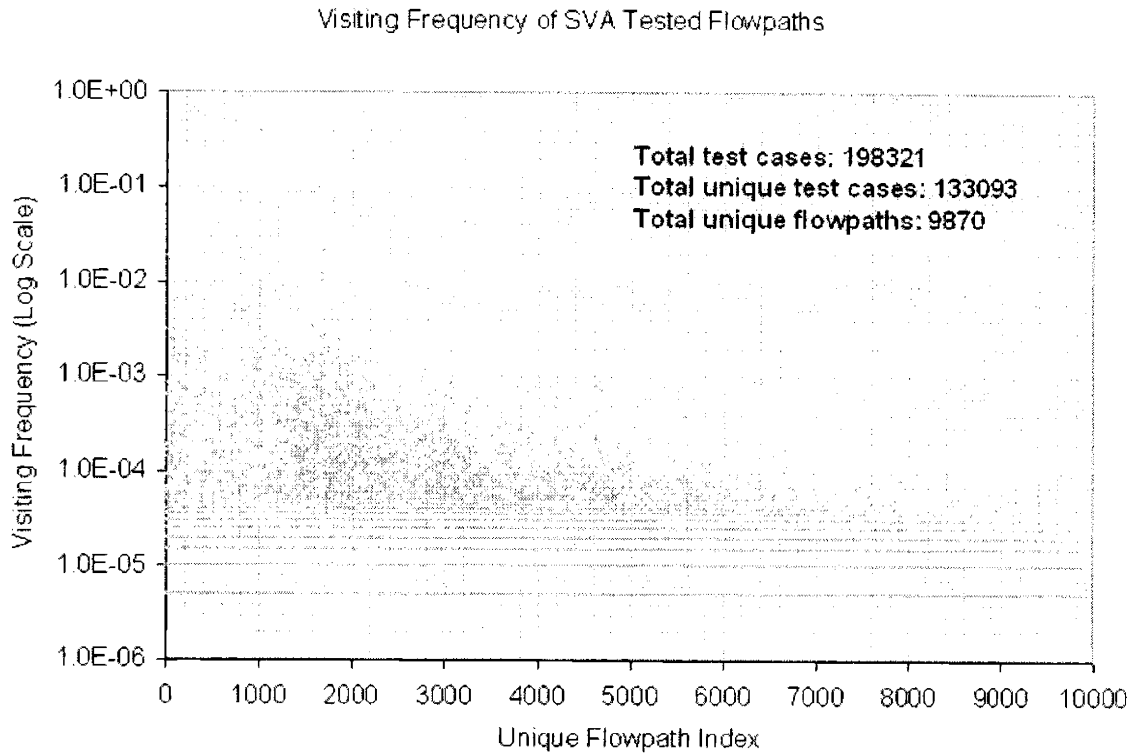


Figure 4-10 Visiting Frequency of Unique 001 SVA Flowpaths

4.6.5.4. 001 SVA Reliability Estimation Based on Feasible Flowpath Coverage

We have collected all the testing data for the 001 SVA program. The reliability estimation method based on software feasible flowpath coverage has been demonstrated for SVA. The unreliability of the tested flowpaths and that of the untested flowpaths are different, and hence are estimated separately. Then the two unreliability values are averaged based on their anticipated probability of being visited during an 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 could be evaluated from a similar program, which in turn, provide more error data points.

12 errors were detected and corrected in the 001 SVA program as a result of 19 the 8,321 test cases. There were 9 type I errors and 3 type II errors. Look at Figure 4-11 for a summary of error data cumulated in the 001 SVA program. For details of the errors found, refer to Appendix 7.8.

Error Overview (Testcases 0.1 ~ Testcases 4999.40)						
FP IDX NO	ERROR TC	Time of Visits	Visiting Frequency	Visits Until Failure		ID X
15	1.4	3	2.25E-05	1		1.4
16	1.5	7	5.26E-05	1		1.5
19	1.10	1	7.51E-06	1		1.10
112	9.8	21	1.58E-04	4		9.4
174	20.11	1	7.51E-06	1		20.11
178	20.16	1	7.51E-06	1		20.16
260	31.14	1	7.51E-06	1		31.14
43	47.14	4	3.01E-05	2		3.9
776	95.18	5	3.76E-05	1		95.18
3457	702.8	1	7.51E-06	1		702.8
6330	2001.1	2	1.50E-05	1		2001.1
5753	2007.2	11	8.26E-05	4		1651.2
<hr/>						
Unique TCs	133093	Error FPs	12	Type I Error		
Total Errors	12	1-Error FPs	9	Type II Error		
Type I Errors	9	2-Error FPs	0			
Type II Errors	3					

Figure 4-11 Errors Detected in SVA 001 Program

There are 9 type I errors and 3 type II errors detected and removed. 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:

$$f_1(p_1) = p_1 f_0(p_1) = C_1 p_1, \quad C_1 = \frac{1}{\int p_1 dp_1} \quad \text{Eq. 4-27}$$

Since the 2nd and 3rd errors are of type I, we get:

$$f_3(p_1) = C_3 p_1^3, \quad C_3 = \frac{1}{\int p_1^3 dp_1} \quad \text{Eq. 4-28}$$

The 4th error is of type II. The distribution is updated as:

$$f_4(p_1) = C'_4 (1-p_1) f_3(p_1) = C_4 (1-p_1) p_1^3, \quad C_4 = \frac{1}{\int (1-p_1) p_1^3 dp_1} \quad \text{Eq. 4-29}$$

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

$$f_{12}(p_1) = C_{12} (1-p_1)^3 p_1^9, \quad C_{12} = \frac{1}{\int (1-p_1)^3 p_1^9 dp_1} \quad \text{Eq. 4-30}$$

The mean value calculated from Eq. 4-30 will be used later to estimate flowpath unreliability. The mean value is:

$$\bar{p}_1 = \int f_{12}(p_1) p_1 dp_1 = \frac{\int (1-p_1)^3 p_1^9 dp_1}{\int (1-p_1)^3 p_1^9 dp_1} = \frac{9+1}{12+2} = \frac{10}{14} \approx 0.7143 \quad \text{Eq. 4-31}$$

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

$$\bar{p}_{II} = 1 - \bar{p}_I = \frac{2}{7} \approx 0.2857 \quad \text{Eq. 4-32}$$

We have $\hat{p}_I = 0.7143$ and $\hat{p}_{II} = 0.2857$.

4.6.5.4.1.2. Probability of encountering Type II Error in 001 SVA

This probability is related only to type II errors. For $p_{f,II}$, the probability that an type II error is encountered during an execution given that there is a type II error on the flowpath. There are only 3 data points for use in this estimation.

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

$$f_1(p_{f,II}) = C_1' (1 - p_{f,II})^3 p_{f,II} f_0(p_{f,II}) = C_1 (1 - p_{f,II})^3 p_{f,II} \quad \text{Eq. 4-33}$$

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

The 2nd and 3rd type II errors are detected during the 2nd and 4th visit respectively. Thus,

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

$$f_3(p_{f,II}) = C_3 (1 - p_{f,II})^7 p_{f,II}^3 \quad \text{Eq. 4-34}$$

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

Now we can obtain the average value for $p_{f,II}$ as:

$$\bar{p}_{f,II} = \int_0^1 f_3(p_{f,II}) p_{f,II} dp_{f,II} = \frac{\int_0^1 (1-p_{f,II})^7 p_{f,II}^4 dp_{f,II}}{\int_0^1 (1-p_{f,II})^7 p_{f,II}^3 dp_{f,II}} = \frac{4}{12} = \frac{1}{3} \approx 0.333 \quad \text{Eq. 4-35}$$

$$\hat{p}_{f,II} = 0.333 \quad \text{Eq. 4-36}$$

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

In this section, we will use a simpler method to estimate unreliability for each tested SVA flowpath as that used in nodal coverage approach. Only the average unreliability value is considered and updated as the test proceeds. For a flowpath that has been tested t times and found to be error-free, the formula to use is:

$$p_e' = \frac{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0}{(1 - \hat{p}_I)(1 - \hat{p}_{f,II})^t p_e^0 + (1 - p_e^0)} \quad \text{Eq. 4-37}$$

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 made to search for a proper prior distribution for p_e^0 turned out to be not very satisfactory. As a result, some rough historic data that were obtained from the U.S. software industry are used in this study. The purpose of presenting this prior distribution here is to give an idea of how to apply realistically this flowpath coverage based reliability methodology.

In Caper Jones' "Applied Software Measure, Assuring Productivity and Quality"[Jone91] published in 1991, valuable software quality data are collected from some 4000 software projects that were developed between 1950 and 1990. The fact that we use data from his report tends to be too conservative because our target software was

developed under the 001 CASE tool and has been through thorough modular testing before this final integrated testing stage, etc. in contrast, 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. This is apparently a more effective means for comparison software applications from different origins. Because we do not want to go into the details of function points, we use the following table, also obtained from [Jone91], to estimate how many function points are within our 001 SVA program.

Size, function points	Effort, staff-month	Schedule, month	Staff	Assignment scope, function points	Production rate, function 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 [Jone91]

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 the 001 SVA program the best: only one person is involved in developing the software and that person spent 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 case with 40 function points, we get a potential defect number of about 100. Given that the 001 SVA program has about 10,000 flowpaths¹⁶, we conclude that about 10,000/100=100 flowpaths have one defect. In other words, 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 number of defects considered into this 0.01 probability is more than might exist within the 001 SVA program and hence gives relatively conservative reliability estimation.

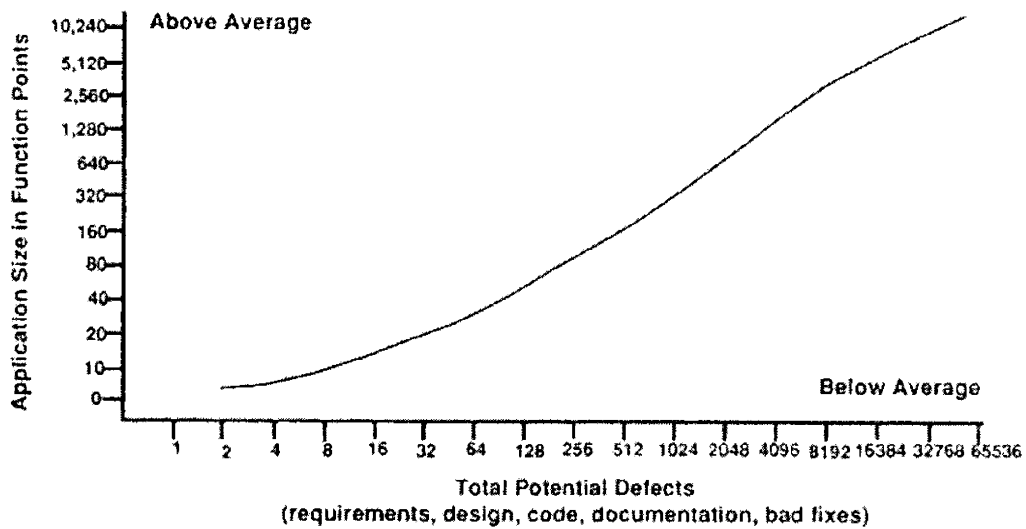


Figure 4-12 U.S. Average Software Defect Potentials

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

$$p_e^0 = 0.01$$

Eq. 4-38

¹⁶ This estimation is described later.

Upon substituting Eq. 4-38 and $\hat{p}_I = 0.7143$, $p_{f,II} = 0.333$ into Eq. 4-37, we get:

$$p'_e = \frac{(1 - 0.7143)(1 - 0.333)^I \times 0.01}{(1 - 0.7143)(1 - 0.333)^I \times 0.01 + (1 - 0.01)} = \frac{0.2857 \times 0.666^I}{0.2857 \times 0.666^I + 99} \quad \text{Eq. 4-39}$$

Eq. 4-39 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:

$$\theta^I = p'_e \times p_{f,II} = 0.333 \times \frac{0.2857 \times 0.666^I \times 0.333}{0.2857 \times 0.666^I + 99} \quad \text{Eq. 4-40}$$

The following graph depicts the unreliability of all the tested 9870 flowpaths that were tested.

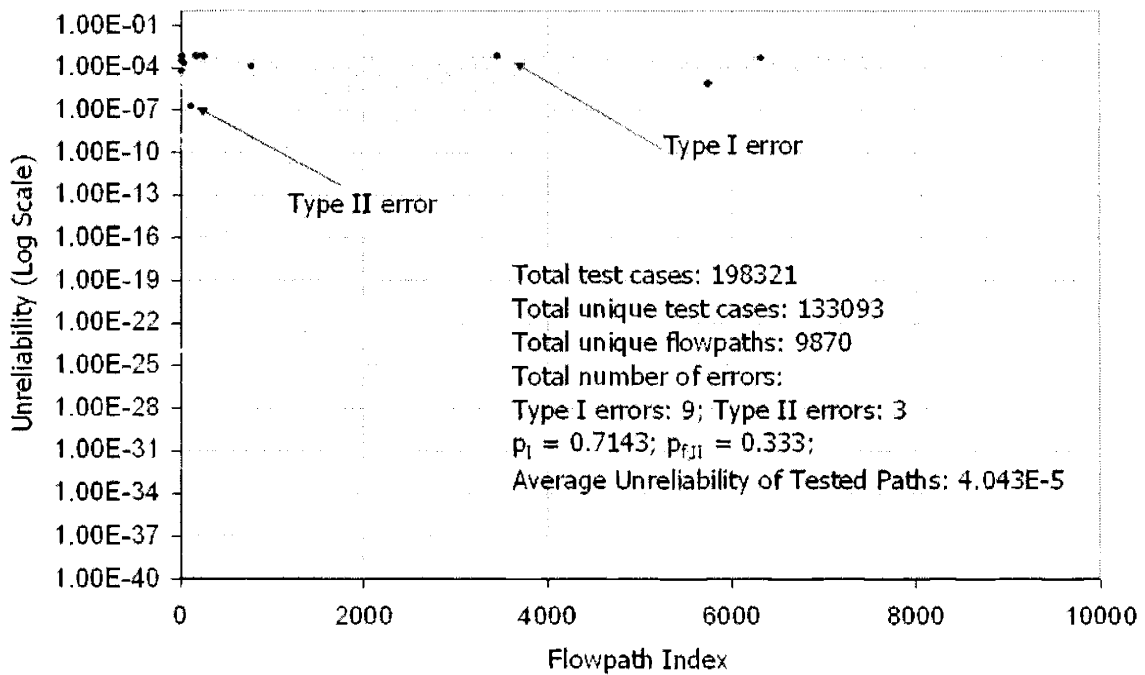


Figure 4-13 Tested Flowpath Unreliability in 001 SVA Program, Average Value

Approach

The visiting frequency plotted in Figure 4-10 is utilized to average the unreliability values that are depicted in Figure 4-13, we get the average unreliability value for the tested flowpath as:

$$\bar{\theta}_{p,T} = 4.043 \times 10^{-5} \quad \text{Eq. 4-41}$$

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

We have explained why there is no theoretical solution to this problem in Chapter3. Two experimental solutions were provided. In this section, results from the two approaches are presented. As we will see, the first approach does not work for 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 determine the total number of existing flowpaths. The first is the simpler and more straightforward. The number of unique flowpath vs. the number of test cases plot is drawn as follows.

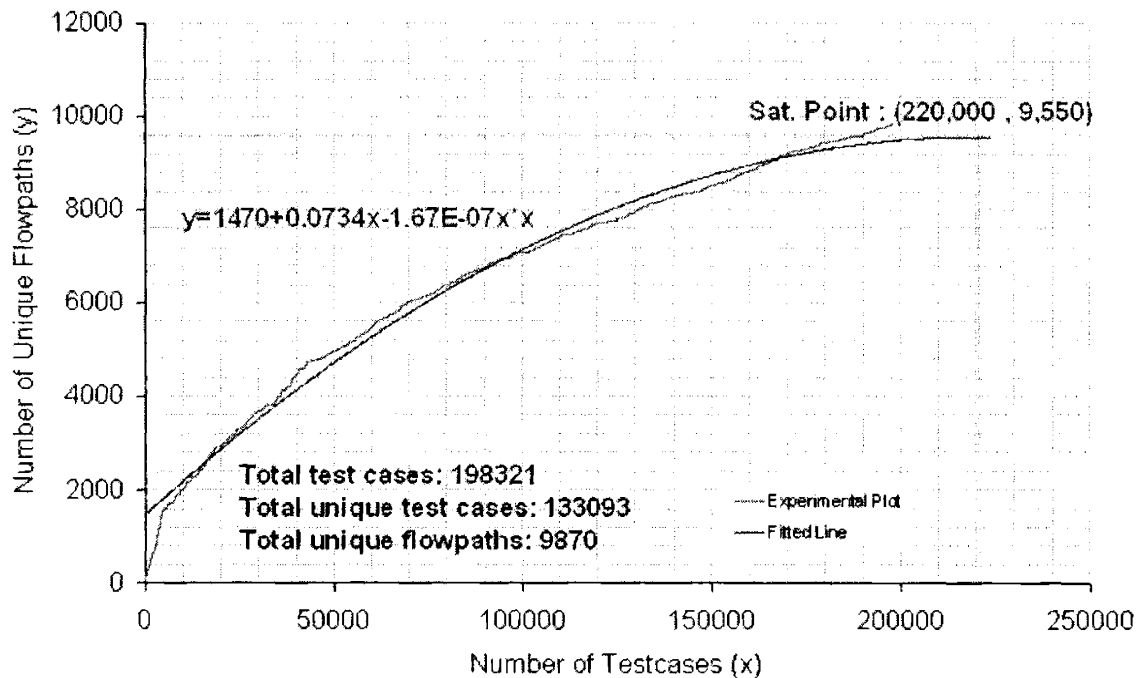


Figure 4-14 Flowpath vs. Test Case of 001 SVA Program

Both the experimental plot and the fitted plot are shown in Figure 4-14. It is evident from the figure that the anticipated total number of cases needed to test every flowpath at least once is smaller than the total number of tests that we have performed on the target program. This is exactly the weakness of this method that we discussed in Chapter 3. When the number of tests carried out is very close to the total number of tests needed to test all flowpaths, the roughness of this approach may result in such a paradoxical outcome.

4.6.5.4.2.2. Estimate from the Second Experimental Solution

This method becomes the only possible solution for this 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

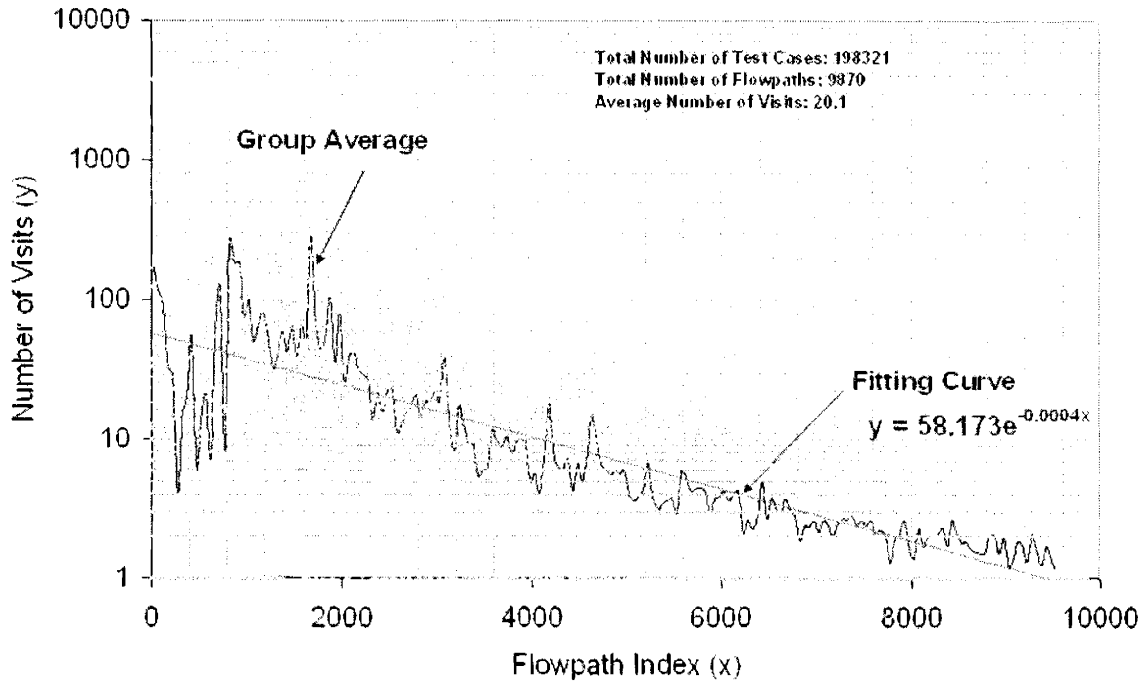


Figure 4-15 Number of Visits vs. Flowpath Index for 001 SVA

According to Figure 4-15, the probability that a flowpath whose index number is among $(x, x + dx)$ will be visited upon the next execution is:

$$P(x)dx = 0.0004e^{-0.0004x} dx \quad \text{Eq. 4-42}$$

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

$$P(X > 9,870) = e^{-0.0004 \times 9,870} \approx 0.01929 \quad \text{Eq. 4-43}$$

Thus, we get:

$$p_{visit,U} = 0.01929, \quad p_{visit,T} = 1 - 0.01929 = 0.98971 \quad \text{Eq. 4-44}$$

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, plots similar to Figure 4-15 are drawn. The probability that a flowpath whose index number is beyond x will be visited upon the next execution always has the form:

$$P(X > x) = e^{-ax} \quad \text{Eq. 4-45}$$

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%	0.7190%	1.9293%
	$P(X > 10000)$	0.6738%	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%	0.2479%	0.8230%
	$P(X > 13000)$	0.1503%	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%	0.0912%	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 throughout the testing process.

4.6.5.4.3. Estimate Unreliability of the Untested Flowpaths

From the estimation result for $p_{visit,T}$ above, we note that the probability that a flowpath that has 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 that we could assume that the observed reliability is the same as the basic reliability. With this on hand, rather than going through the complicated process of estimating the unreliability

of the software, we can take a much shorter approach. The unreliability of the untested flowpaths can be approximated by the observed unreliability of the tested flowpaths without removal of the detected defects. More specifically, we set the unreliability of the flowpaths that have type I errors on them to unity; and that 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 is 2.2023E-04.

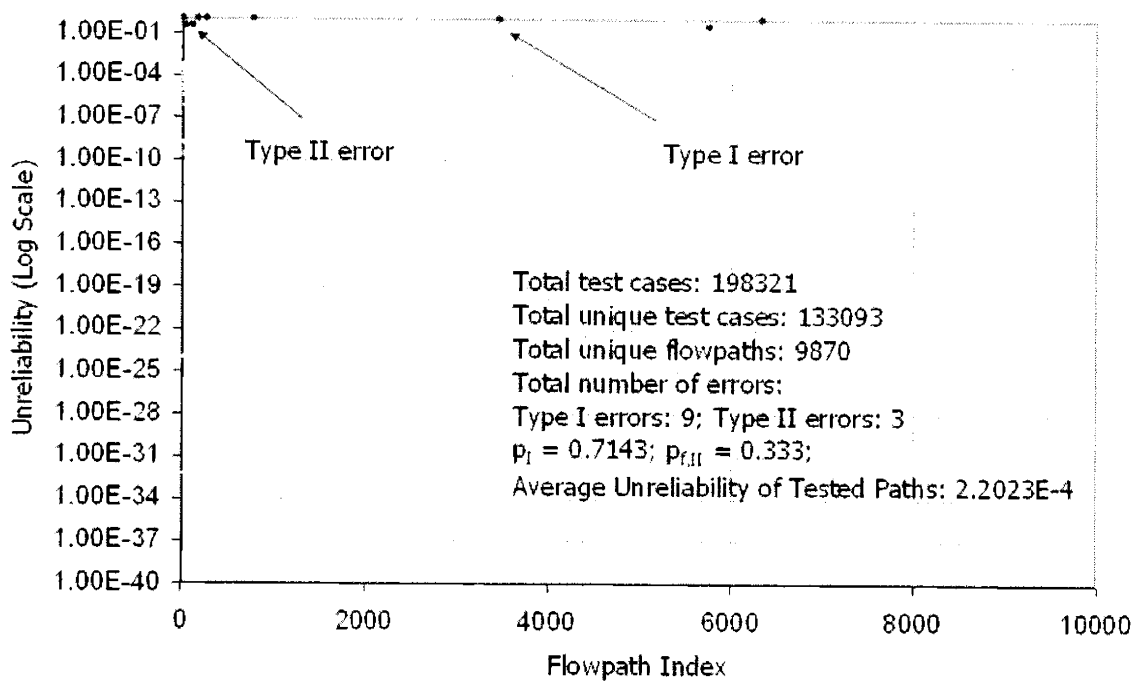


Figure 4-16 Unreliability of Untested Flowpaths by Average Approach

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 combined to a weighted average with their respective probabilities of being visited during an execution as the weights. The total unreliability is:

$$\theta_p = P_{visit,T} \bar{\theta}_{p,T} + P_{visit,U} \bar{\theta}_{p,U} \quad \text{Eq. 4-46}$$

The data 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%	4.043E-5	1.929%	2.2023E-4

Table 4-5 Unreliability Data from Flowpath Coverage Based Approaches

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

$$\theta_p = 98.071\% \times 4.043 \times 10^{-5} + 1.929\% \times 2.2023 \times 10^{-4} = 4.3895 \times 10^{-5} \quad \text{Eq. 4-47}$$

The corresponding reliability value is:

$$R_p = 1 - \theta_p = 1 - 4.3895 \times 10^{-5} = 99.99561\% \quad \text{Eq. 4-48}$$

4.6.6. Results Summary

Manually and automatically (by means of random sampling), we have performed 198,321 tests on the 001 SVA program. Among these tests, 133,093 unique input data sets were used. 12 errors were detected and removed during the process, among them there were 9 type I errors and 3 type II errors.

432 SVA (non-001-built-in nodes) nodes were identified in the program. With a uniform prior distribution, the average nodal unreliability of SVA nodes is $\theta_{n,SVA} = 2.2360 \times 10^{-3}$ and the overall SVA unreliability from nodal coverage approach is

$\theta_n = 1.5886 \times 10^{-3}$. By comparing nodal coverage based approach and flowpath coverage based approach, we can get an approximate value by combining the nodal coverage based estimation result with the prior average unreliability value and get $\bar{\theta}_{p,Pe_0} \cong 1.5886 \times 10^{-5}$, which gives a reliability value of $R_{p,Pe_0} \cong 99.9984\%$.

9870 unique flowpaths were recognized on the 001 SVA program. The probability that an error belongs to the type I group is $p_I = 0.7143$ and that it belongs to type II group is $p_{II} = 0.2857$. 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 the 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 $\theta_{p,T} = 4.043 \times 10^{-5}$ and that of the untested flowpaths is $\theta_{p,U} = 2.2024 \times 10^{-4}$. The overall unreliability value is $\theta_p = 4.390 \times 10^{-5}$ with the corresponding reliability value $R_p = 99.9956\%$. This is very close to the value obtained from nodal coverage approach, $R_{p,Pe_0} \cong 99.9984\%$.

	Uniform prior distribution	0.01 average prior Unreliability
Nodal Coverage	1.589E-3	1.589E-5
Flowpath Coverage	-	4.390E-5

Table 4-6 Unreliability Estimation Results for the 001 SVA Program

5. Discussions

From the SVA testing and reliability estimation results, we have observed several interesting features of this type of software. They are presented and discussed in this chapter.

The plot of flowpath indices versus their number of visits during the random testing process is utilized to estimate the probability that a certain flowpath will be executed upon demand. The observation that leads us to this experimental technique is that the earlier a flowpath is identified during the random testing process, the more visits it receives during the random process. There is a linear relationship between the flowpath index and the logarithm of its number of visits. Apparently, this is very intuitive because the higher the probability that a flowpath will be visited, the earlier it should be identified. As was mentioned earlier, the probability density functions that have been used in this study to sample input data randomly are normal distribution and uniform distribution. In contrast, the numbers of visit to different flowpaths are very different. The majority of the dispersed input data were mapped to a small group of flowpaths. This was not a coincidence. As in many other software systems, the nodes and flowpaths of SVA can be divided into two parts depending on their functions. There is always one part of the software that is dealing with routine process tasks and another part that is built to handle various abnormal situations. In SVA, the routine processes include averaging sensor readings, sensor reading deviation checking against the average, sensor status checking, sensor status marking, sensor range checking, asking the operator for inputs, etc. The abnormal situations include wrong operator input, wrong sensor reading range, all-bad-sensors, etc. Obviously, the first part of the program is dealing with the majority of the input data while the second part of the program which tends to have longer code thanks to the various abnormal scenarios, is dealing with far less input data. For this reason, the majority of the input data hit a relatively small group of flowpaths (the routine task handling flowpaths) in the software.

One of the two most important criteria used when we chose our testing method was the completeness of the method. The safety-critical nature of the software that we

are studying requires we test the software as completely as possible. Given this criteria, a reasonable question to ask is how complete is complete enough. Is there a point at which we are one hundred percent sure that we have tested the software completely? Theoretically, such a point can only be reached from a black box point of view. That is, it is obtained when we completely tested every possible input data set without any failures. The reality is that we never know what the complete data universe is. From the white box point of view, there is no such a thing as complete situation. First, we do not know the complete set of possible (input data reachable) flowpaths in the software. Second, even if we have tested every possible flowpath in the system, we cannot guarantee they will not cause a failure from the fact that they have not caused any failure corresponding to the tests that have been applied to them. This can be seen from the definition of type II error. No matter how many tests (t) have been performed on a flowpath, there is always a non-zero probability $((1 - p_{f,II})^t \neq 0)$ that there is an undiscovered type II error left on it. As a practical matter, this may be not true because once we have applied all the possible input data sets that can lead to execution of this flowpath; the probability is equal to zero. But because of the limitation of our knowledge, we can never say there is no error on a flowpath, which leads to the introduction of type II error. In our study, no error from any of the flowpaths on the 001 SVA program was found after 4 visits. To conclude, although, we never say that we have completely tested a software system, the probability of an error still existing on a flowpath that has been tested t times (in SVA, 4 is a typical value of t) is very small.

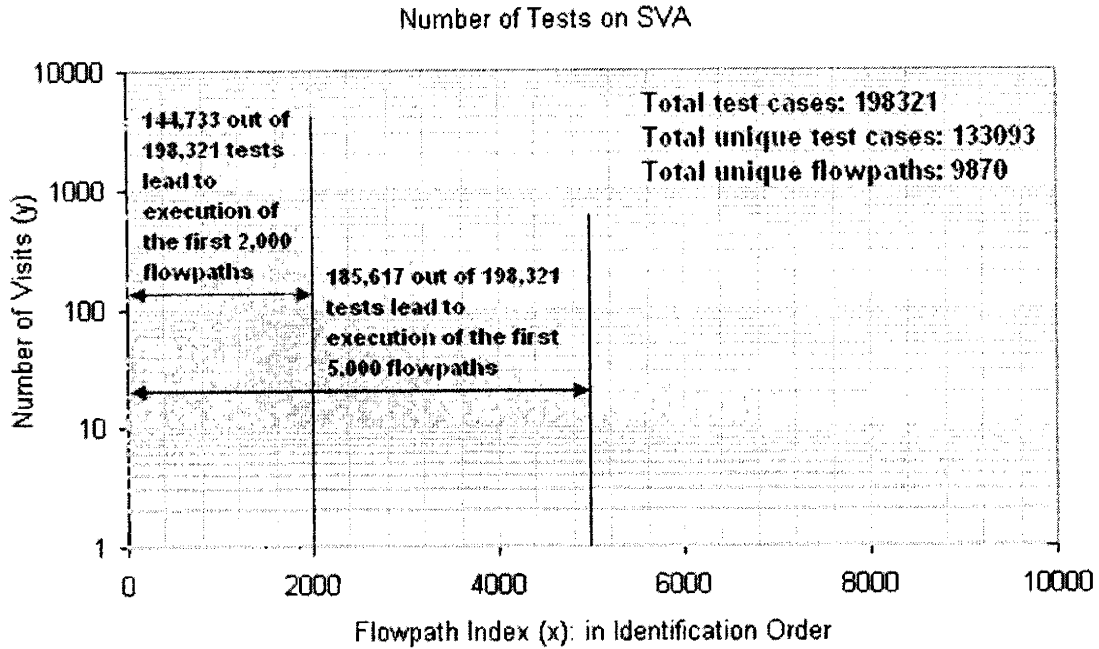
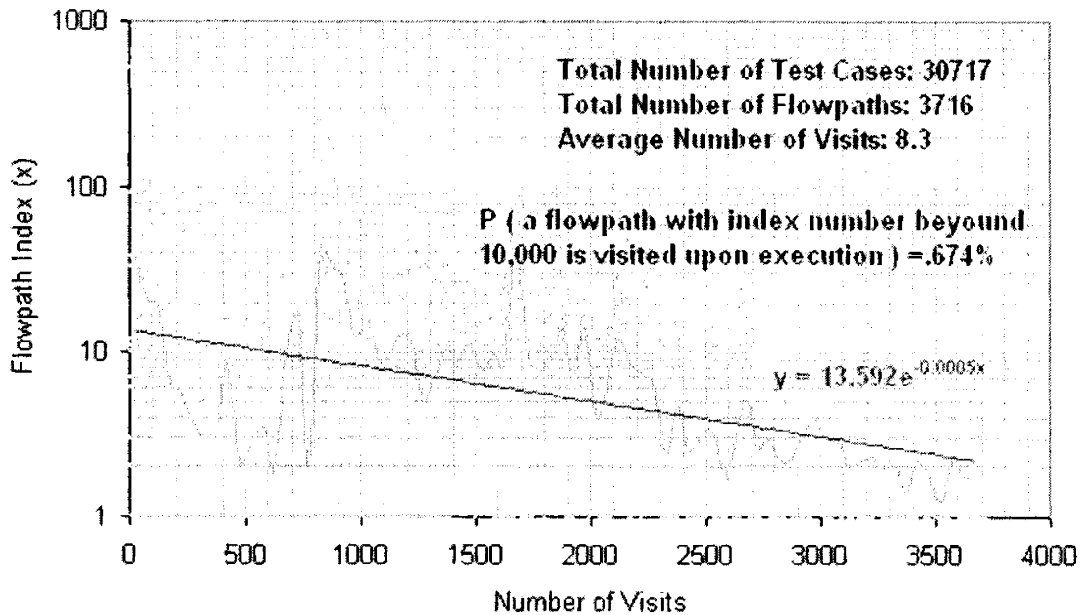


Figure 5-1 Uneven Flowpath Hitting Rates

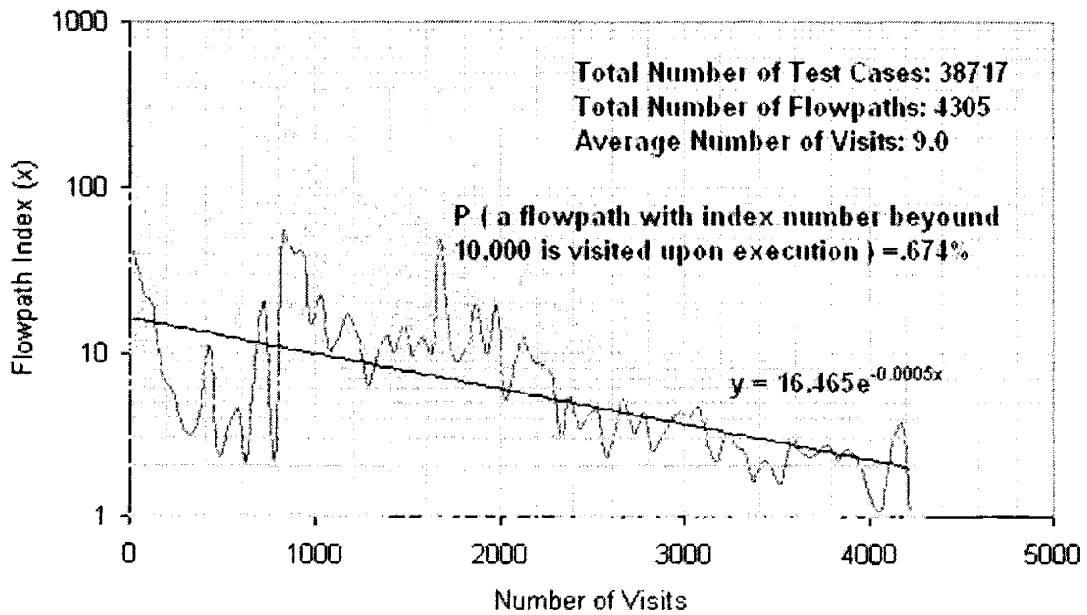
The experimental method used to estimate the probability that an untested flowpath will be visited during the next trial is an important tool that was developed in this study. Thus, it is imperative to show the consistency of its estimating results. The probability that a flowpath numbered 10,000 will be executed on the next trial is estimated from this method at different testing stages. As can be seen from Figure 5-2, the estimated probability that a flowpath numbered 10,000 will be visited upon next execution is consistently 0.674% based on information obtained from 30,717 tests, 38,717 tests, 58,717 tests, 78,717 tests, 118,321 tests, and 158,321 tests. Based on data from 198,321 tests, we obtained an estimate, 1.832%, that is greater because of the long and persistent tail formed by the last few identified flowpaths. Consider the following. First, the percentage that we are estimating is very small. Thus, it will not affect the overall reliability estimation very much. Second, the exponential distribution is asymptotically decreasing to zero as the total number of flowpaths increases to infinity. But the smallest number of visits to a flowpath, if it has been tested, is one. Thus, the exponential function obtained is weighted more towards its tail. Refer to Figure 5-3 for

this effect. This makes the overall reliability estimate more conservative because it makes the visiting probability of an untested flowpath greater.

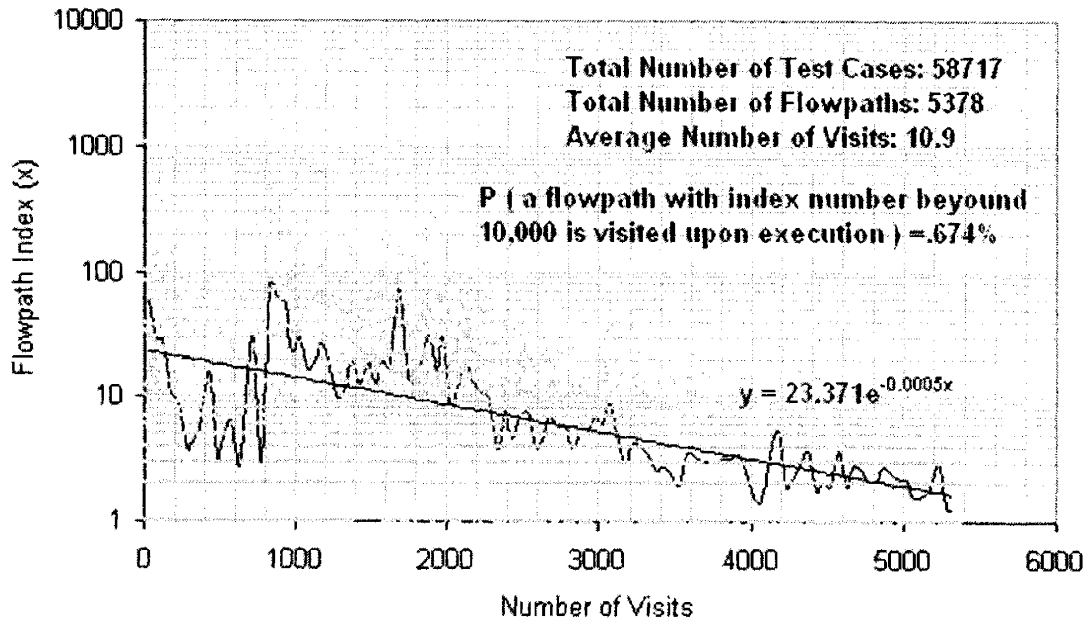
Number of Visits to Flowpath (0 ~ 799), All Test Cases



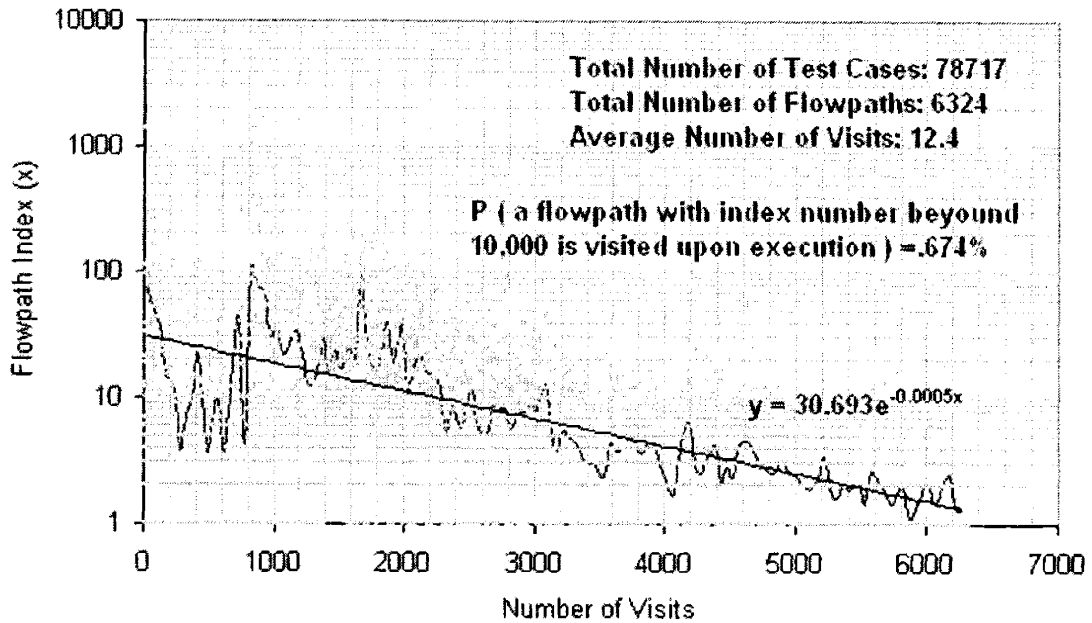
Number of Visits to Flowpath (0 ~ 999), All Test Cases



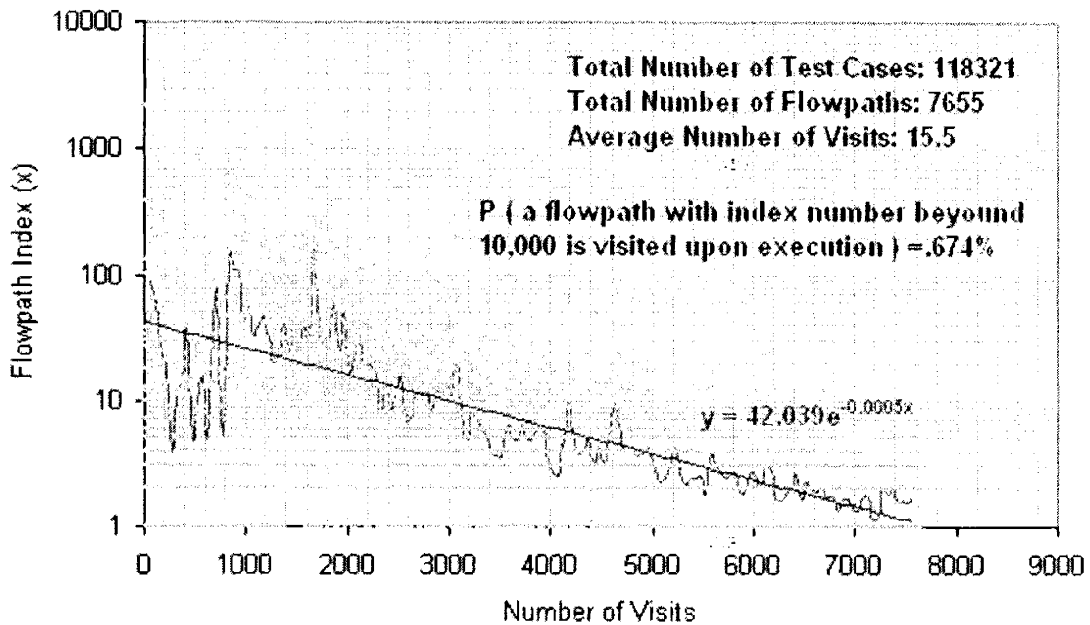
Number of Visits to Flowpath (0 ~ 1499), All Test Cases



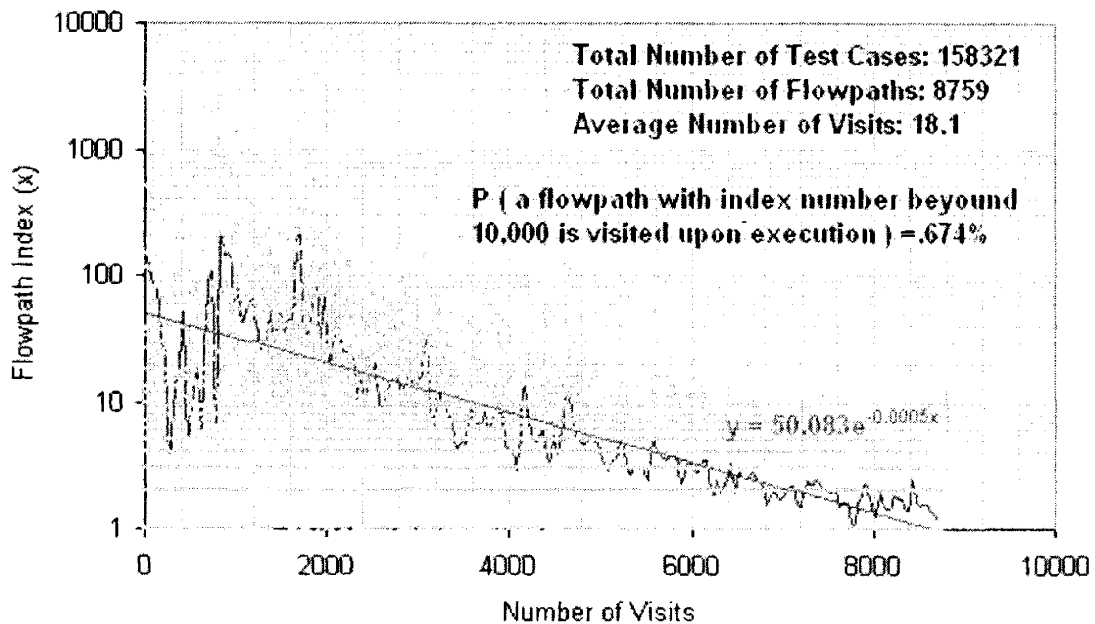
Number of Visits to Flowpath (0 ~ 1999), All Test Cases



Number of Visits to Flowpath (0 ~ 2999), All Test Cases



Number of Visits to Flowpath (0 ~ 3999), All Test Cases



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

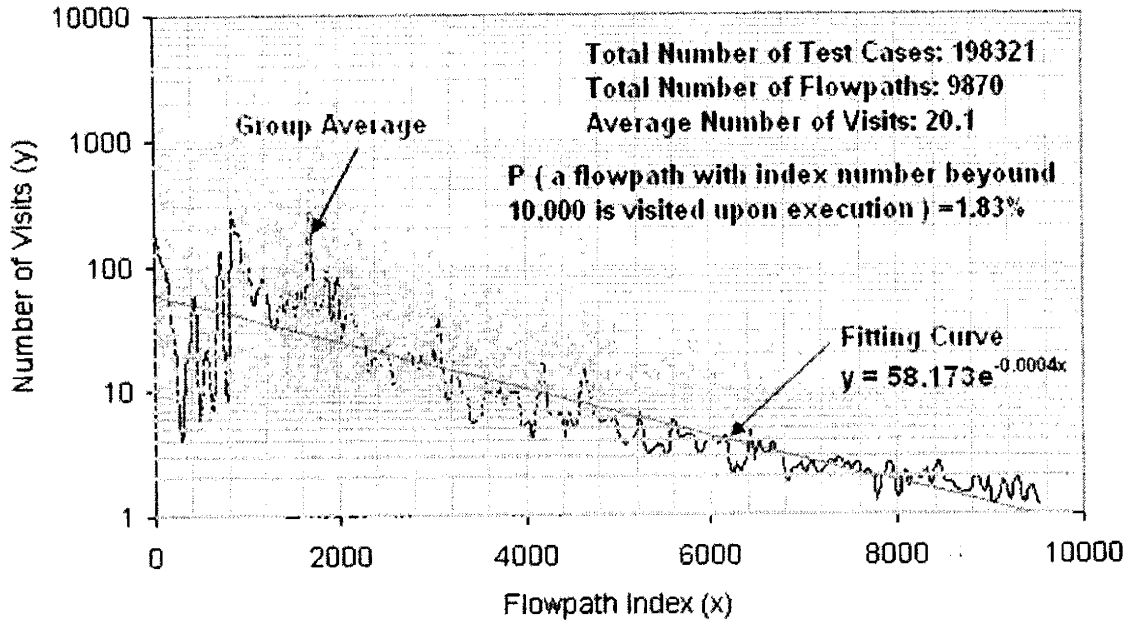


Figure 5-2 Consistency of Flowpath Visiting Probability Estimation

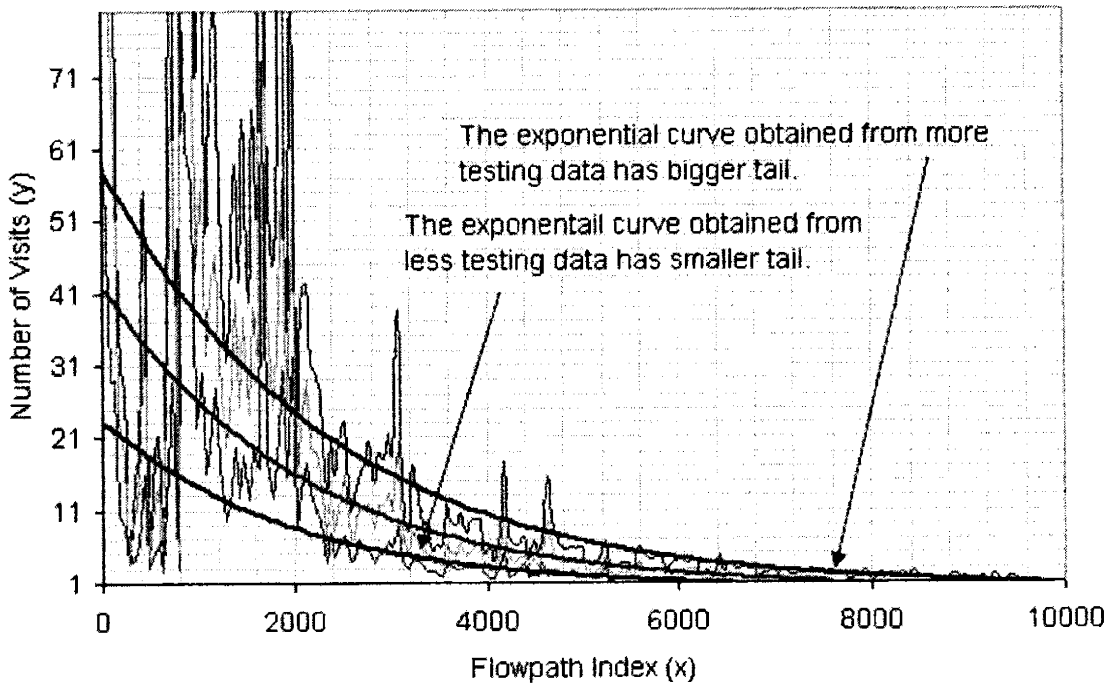


Figure 5-3 Bigger Tail of Exponential Function Estimated from More Testing Data

6. Conclusions and Future Work

6.1. Conclusion

There are two major tasks in this project. Both were driven by the safety-critical feature of the software system in nuclear industry. They are: (1) Improve software quality — develop a systematic method to test the software as complete as possible; and (2) Assess software quality — develop a standard approach to quantify software reliability. They are two elements of the last stage of the formal software development cycle — verification. There is always one constraint that has played an important role in all of the methodology development process, i.e., feasibility. However sound an approach is theoretically, it has to be automated in order to be rendered of real use. For this reason, we have been working on the design of the methodology itself and demonstrating every step on a typical nuclear safety-critical software application, SVA, at the same time.

Grey box testing method is initiated. It not only takes advantage of the knowledge that we have about the internal structure of given piece of software and but also overcomes the difficulty of identifying input data that can activate specific structures in the software. The probability density distributions of the input variables are obtained from the operational profile. A Monte Carlo method is utilized to generate software input data. Flowpaths that are visited are identified and recorded, and this information is in turn used to judge the coverage speed and percentage of the testing process. A software tool is developed to examine the percentage of nodes in the software under question that have been tested at any stage during testing. It was contrary to previous claims that the Monte Carlo sampling was too slow in terms of testing coverage speed, it was shown to have a very reasonable flowpath discovery efficiency thanks to the simplicity of the nuclear-related software. Furthermore, the main purpose of testing the software is to detect all the errors in the most reachable flowpaths in the software. Thus it is essential to make sure that the flowpaths with the highest probability to be visited during real operating situation are the ones that have been tested the most thoroughly. This is

preferable to achieving a uniform coverage speed on every flowpath of the software. In this approach, we also required that all the nodes in the safety-critical software be tested at least once. Our experiment has shown that those nodes that are very hard to reach tend to have a higher probability to contain errors (though this is not the case for flowpaths). And they may very possibly be surplus nodes that cannot be reached by any input data.

Two Bayesian-based white box reliability quantification methods were proposed and demonstrated in this work. The nodal coverage based method is the more approximate but is much quicker and easier to achieve. The flowpath coverage based method is more precise and thus give us more insights into the special features of the target software. Both methods have the great advantage of Bayesian methods, i.e., they can systematically include any subjective and objective information of the given software at any stage.

In developing the flowpath coverage based method, we divided the software errors into two classes, type I and type II. This helped us achieve a better knowledge about reliability of the software. Type I errors are very easy to be found because they can be detected during any visit to their host flowpaths. In contrast, there is always a small possibility for a type II error to exist on a flowpath regardless of how many times the host flowpath has been tested. Because of the existence of this error type, we can never claim that a piece of software can never fail (except for the special case where we are able to complete an exhaustive black box testing, by examining every output corresponding to every possible input of the give software).

Because of the impossibility of theoretically calculating the total number of reachable flowpaths in a piece of software, an experimental method was created and implemented to estimate the probability that an untested flowpath will be visited upon next execution of the software. After some research in this area, we noted that it was not important to know how many flowpaths have been tested and how many have not been tested. The essential thing to know is how important those untested flowpaths are, which, in this project is expressed as the probability that each of them is visited during any execution of the software.

There is no obvious information upon which we can rely to obtain estimates of an untested flowpath. Our approach is always to start from what we know, bridging the gap

between knowns and unknowns with their similarities and correcting the inaccuracies at the end.

As pointed out in previous work, it is essential to automate the proposed methodology. To do so was a major hurdle for this research. At this point, this concern has been mainly resolved. All the methodologies discussed in this thesis have been automated and implemented on the sample nuclear safety-critical software, whose level of complexity is typical of the nuclear industry.

The work in this research has provided a framework for thorough software testing and systematic reliability quantification. The method has been demonstrated on a nuclear safety-critical software program and it is applicable to all testable [Sui98] software projects.

6.2. Future Work

Future work could proceed in a variety of directions. This section is written to present ideas for potential directions of future work.

In this work, the flowpath coverage state is a function of the probability density function (PDF) of the input data. The estimated reliability value is specific to the PDF that has been chosen for input data sampling process. Under no circumstances, can we have a PDF which precisely reflects the most likely service pattern of the given software. Thus, it is very important to know the sensitivity of the resulting flowpath coverage to different input PDFs. We want to know how good the reliability estimation result is when some amount of error or some amount of uncertainty exists in the PDF. From the experience obtained from the experimental work on the sample software, it is very possible that PDFs that differ significantly from one another may give very similar flowpath coverage results. The possible reason could be that flowpaths that deal with normal operating situations always have a higher probability of being visited while flowpaths designed to handle abnormal situations have a much less visiting probability. If this sensitivity feature could be captured quantitatively, or even qualitatively, the tester will have a better idea of how precise the PDF is required for a reasonable reliability estimation result.

The Monte Carlo method was chosen for its simplicity and manageable effort. But it is believed that there are drawbacks to this method. The most significant one was its slow flowpath coverage rate because it may provide a large amount of input data that concentrates on some portion of the given software. If the underlying PDF is very close to the real situation, it is truly the feature that we want. As described in previous chapters of this thesis, because of the existence of type II errors, it is never proper to say there is no error on a flowpath. Thus, we do want the more frequently visited flowpaths to be tested more thoroughly. But it is also important that those not-so-frequently visited flowpaths be covered during the testing with a manageable amount of testing effort made. Some flowpath coverage efficiency analysis of the Monte Carlo technique is performed to address this need.

One important input to the Bayesian updating method is the prior distribution function of the parameter in question. A good prior distribution function can significantly increase the speed of result convergence. In this work, because of the lack of prior knowledge of given software system, a simple average value was used to serve as a prior distribution. More effort is needed to search for better prior reliability distribution functions.

While demonstrating the methodology on SVA in the research work, the probabilities associated with type I and type II error were uniformly estimated from the error information itself. Because we had a very small amount of information from the software (the error in the software developed according to a formal development process has very small number of errors even before any integrated test has been done on it), the uncertainty was relatively large. The only way to decrease such uncertainty is to provide more data. It would be very helpful if error data from similar software systems could be accumulated and applied to the Bayesian model to update the probabilities associated with type I and type II errors.

As described earlier, the shielding effect concept was introduced to address the difference between tested and untested portions of the software in order to estimate the reliability of the untested flowpaths from the information obtained from the tested flowpaths. Intuition suggests that the shielding effect decreases exponentially as the total number of flowpaths that have been tested when an error happens. This conclusion is

only supported in this research by a very limited amount of experimental data. This effect can and needs to be studied in more detail. The approach is to seed more errors that have been removed back into the software one at a time and observe the total number of tests that will fail because of that error. Experimental data could also be obtained from similar software systems as well.

A very strong and important assumption used in this research was the availability of an existed oracle. In reality this is not true. More research needs to be done to develop an automatic mechanism to provide proper oracle. Because of the simplicity and relatively strict and clear logic of the instrumentation control software system under study, this is certainly a doable task.

In order to show the generality of the method and of all the automation software tools developed during this study, they need to be applied to more safety-critical software systems. The reliability values obtained at similar testing stages of similar software systems should be compared to measure the standardness of the methodology.

7. Appendix

7.1. Arithmetic Average Vs. Geometric Average

Definition of arithmetic average:

$$\bar{a} = \frac{w_1^a a_1 + w_2^a a_2 + \dots + w_n^a a_n}{w_1^a + w_2^a + \dots + w_n^a} \quad \text{Eq. 7-1}$$

Definition of geometric average:

$$\bar{g} = \sqrt{(w_1^g + w_2^g + \dots + w_n^g)} g_1^{w_1^g} \times g_2^{w_2^g} \times \dots \times g_n^{w_n^g} \quad \text{Eq. 7-2}$$

How do we decide 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 multiply 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 an 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:

$$\theta = R_1 \times R_2 \times \dots \times R_n \quad \text{Eq. 7-3}$$

Expressed in unreliability, Eq. 7-3 becomes:

$$\theta = 1 - R = 1 - (1 - \theta_1)(1 - \theta_2) \dots (1 - \theta_n) \quad \text{Eq. 7-4}$$

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

$$\theta \approx \theta_1 + \theta_2 + \dots + \theta_n \quad \text{Eq. 7-5}$$

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 roughly¹⁷ 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 the arithmetic average and geometric average, the arithmetic average is apparently the proper formula that we should use to calculate the software unreliability based on its nodal coverage.

¹⁷ "Roughly" is because we assume an independent relationship among the nodes on each path, which is not true in reality.

7.2. 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”, which means 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 readings¹⁸ (s_1, s_2, \dots, s_n);
- Values of PAMI sensor readings¹⁹ (p_1, p_2, \dots, p_m);
- Lower boundary for narrow range (lb);
- Upper boundary for narrow range (ub);
- Expected variation for normal sensors (ev);
- Instrumental uncertainty for normal sensors (iu);

¹⁸ How many values for normal sensor readings are needed for one input set depends on the “Total number of normal sensors” specified in “GS_VALID.H”.

¹⁹ How many values for PAMI sensor readings are needed for one input set depends on the “Total number of PAMI sensors” specified in “GS_VALID.H”.

- 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);

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.

7.3. 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 7-1 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 7-2 Oracle SVA Program Output Screen

7.4. 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;

7.5. 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 status²⁰;
- 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;
- 1 : Valid_pami;
- 0 : pami_operator_permissive;
- 0 : operator_select;
- 0 : selected_sensor_by_op;
- 0 : fault_select;

²⁰ Sensor status recording notations: 1 for GOOD, 2 for BAD, 3 for SUSPECT, 4 for DEVIANT.

- 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;

7.6. TMaps Constructed for SVA 001 Input Data

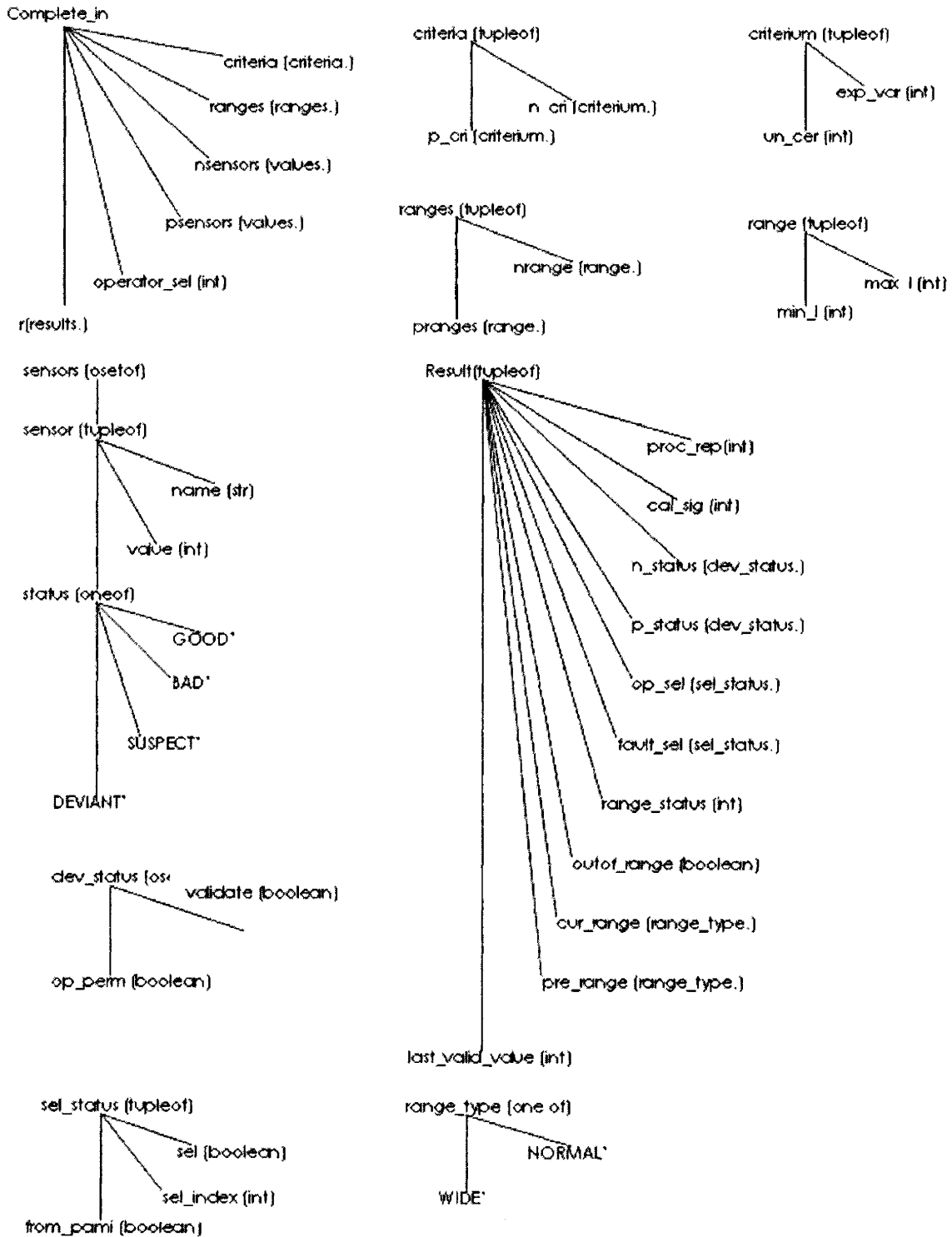


Figure 7-3 TMaps Storing Input Data for 001 SVA Program

7.7. 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 ← 2nd. 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
```

7.8. SVA Error Details

TypeI Error	
TypeII Error	
Node Deleted	
Error Details	Test Cases 0.1 ~ 4999.40
Error TC	1.4
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->valid_sig_from_psensors_or_not-->calculate_sig_from_psensors-->pami_devcheck_fail-->unsatisfied_deviation1-->max_dev
FP IDX NO	15
IDX	1.4
Number of Visits Until Failure	1
Error Description	In "max_dev" of OO1 program, the conditions used to find the maximum deviation are wrong.
Correction	In "max_dev", use the following conditions to find the maximum variation from the average: For each sensor value, [current variation] = [current sensor value] - [average] . If [current variation] > [current maximum variation], mark current sensor as the sensor with the current greatest variation. If [current variation] == [current maximum variation], compare variations of these two sensors (current sensor and the sensor currently recorded as the sensor with maximum variation) from [last_valid_signal], mark the sensor with greater variation as the sensor with current maximum variation.
Error Type	Ambiguity in Specification: unexpected details that not defined in the specification, hence interpreted differently in different implementations.
Error TC	1.5
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->fault_select-->failed_validation-->op_sel_is_false
FP IDX NO	16
IDX	1.5
Number of Visits Until Failure	1
Error Description	In the oracle program, [result.pre_range] could be modified only when the process representation is the average of current good sensors (which means the average has satisfied the deviation check), or, the operator has selected a sensor value as the process representation. In another word, [result.pre_range] is only modified when [satisfied_deviation] or [op_sel_is_true] is true. In OO1 program, [result.pre_range] is modified even when the process representation is selected by fault. Note: [result.pre_range] is the variable whose value decides whether or not [average] is declared as "Out of Range in Normal Range".
Correction	In "op_sel_is_false", do not change [result.pre_range].
Error Type	Ambiguity in Specification: unexpected details not defined in the specification, hence interpreted differently in different implementations.

Error TC	1.10
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->fault_select-->failed_validation-->f_sel_from_pami-->last_valid_check2
FP IDX NO	19
IDX	1.10
Number of Visits Until Failure	1
Error Description	In "last_valid_check2" of OO1 program, when trying to find a representative value from PAMI sensors, the value of last [fault_selected] sensor should be used instead of [last_valid_signal], i.e. find the PAMI sensor whose value minus the value of last [fault_selected] sensor value is the minimum.
Correction	In "last_valid_check2", choose the PAMI sensor value that is closest to value of last [fault_selected] sensor instead of closet to value of [last_valid_signal] as the representative value.
Error Type	Ambiguity in Specification / Misunderstanding of the Specification Logic
Error TC	9.8
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->valid_sig_from_nsensors1-->valid_sig_from_nsensors-->devcheck_satisfy-->range_check-->high_limit_check-->calculate_margin-->margin
FP IDX NO	112
IDX	9.4
Number of Visits Until Failure	4
Error Description	This mismatch between oracle output and OO1 program output happens during range check of the normal value average is done. In the <u>oracle</u> program, since [range_status] is "2", "low_limit_check1()" is called. It says: if (value >= normal_min_range+(normal_max_range-normal_min_range)*0.02) (A) return (IN_RANGE); else return (OUT_OF_RANGE); In <u>testcase9.8</u> : int value=204 // [value] is the average of current normal sensor readings int normal_min_range = 200 int normal_max_range = 400 (A) becomes if (204 >= 200+(400-200)*0.02) in oracle (developed with C), which is equivalent to if ((int 204) >= (float 204.0000000))<=>if (FALSE), yielding the result of (OUT_OF_RANGE). In "margin" and "calculate_margin" of OO1 program, the above comparisons are made between ints. This means: if (204 >= 200+(400-200)*0.02) is equivalent to if ((int 204) >= (int 204))<=>if (TRUE), yielding the result of (IN_RANGE).
Correction	In "margin" and "calculate_margin" of OO1 program, use "rat" (rational number) instead of "int" to make comparasons. But in OO1, value "204.0" is still equal to "204". Thus, we made one more modification within "margin". We added "0.00001" to the calculated margin, i.e. make the margin a little larger to achieve the same result as in the oracle program.
Error Type	Different computer languages or software development tools yield different results when silent data type conversions are made.
Why is it a type II	The above phenomenon happens only when the boundary condition is met. Whenever the input data are chosen so that the following satisfied: [value] >= (normal_min_range+(normal_max_range-normal_min_range)*0.02), it is possible that exactly the same flowpath is executed. But only the input data have been selected so that [value] == (normal_min_range+(normal_max_range-normal_min_range)*0.02), we can find the above phenomenon. The error can only be detected when the boundary condition is met.

Error TC	20.11
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->output-->output_op_and_range-->
FP IDX NO	174
IDX	20.11
Number of Visits Until Failure	1
Error Description	This is a format difference. In the <u>oracle</u> program, when the operator has selected a sensor, its name is displayed. In the <u>OO1</u> program, when the operator has selected a sensor, its value is displayed.
Correction	In "output_op_and_range", change "op_str=merge:str("<",op_str_label,value_str)" to "op_str=merge:str("<",op_str_label,name_str)". Add a new fmap "fault_select_name" in OO1 program to find the corresponding sensor name.
Error Type	Ambiguity in Specification: the output format is not described in the specification.
Error TC	20.16
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->after_op_fault_sel1-->after_op_fault_sel
FP IDX NO	178
IDX	20.16
Number of Visits Until Failure	1
Error Description	This is a coding error.
Correction	In "after_op_fault_sel" of OO1 program, [result.cur_range] should be modified according to [result.op_sel.from_pami] rather than [result.op_sel.sel].
Error Type	Carelessness of the programmer.
Error TC	31.14
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->fault_select-->failed_validation-->op_sel_is_false
FP IDX NO	260
IDX	31.14
Number of Visits Until Failure	1
Error Description	In the <u>oracle</u> program, [result.pre_range] could be modified only when the process representation is the average of current good sensors (which means the average has satisfied the deviation check), or, the operator has selected a sensor value as the process representation. In another word, [result.pre_range] is only modified when [satisfied_deviation] or [op_sel_is_true] is true. In <u>OO1</u> program, [result.pre_range] is modified even when the process representation is selected by fault. Note: [result.pre_range] is the variable whose value decides whether or not [average] is declared as "Out of Range in Normal Range".
Correction	In "op_sel_is_false", do not change [result.pre_range].
Error Type	Ambiguity in Specification: unexpected details not defined in the specification, hence interpreted differently in different implementations.
Note	This node has been deleted from the OO1 program later in order to achieve automatic random operator select. This error information won't be used to do reliability estimation.

Error TC	47.14
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->valid_sig_from_psensors_or_not-->calculate_sig_from_psensors
FP IDX NO	43
IDX	3.9
Number of Visits Until Failure	3
Error Description	In "calculate_sig_from_psensors" of OO1 program, forget to modify [result.p_status]
Correction	In "calculate_sig_from_psensors" of OO1 program, add fmap "mod_p_status".
Error Type	Carelessness of the programmer.
Why is it a type II	This program runs continuously. The status of every variable stays the same as in the previous run if no modification is made. If the program forgets to modify status of some variable and the value of that variable left from last run happens to be correct for current run, this error cannot be found. Only when [result.p_status] should be different from last run, this error could be noticed.

Error TC	95.18
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RenMenu_s-->SVA_process_s-->fault_select-->failed_validation-->f_sel_from_pami-->last_valid_check2
FP IDX NO	776
IDX	95.18
Number of Visits Until Failure	1
Error Description	In "last_valid_check2" of OO1 program, ignored the fact that the initial value of [mindev] was set to -1.
Correction	In "last_valid_check2" of OO1 program, ad one more condition so that the deviation of the first sensor is always given to [mindev] for futher comparison.
Error Type	Programer's ingorance of a coding detail (the initial value of some variable). <NODAL ERROR>

Error TC	702.8
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RenMenu_s-->SVA_process_s-->valid_sig_from_nsensors1-->valid_sig_from_nsensors-->devcheck_satisfy-->range_check-->high_limit_check
FP IDX NO	3436
IDX	702.8
Number of Visits Until Failure	1
Error Description	In "high_limit_check" of OO1 program, [fault_selected_value] should be compared with [high_limit] - ([high_limit] - [low_limit]) * 0.02. But before modification, the program compared [fault_selected_value] with [high_limit] + ([high_limit] - [low_limit]) * 0.02.
Correction	In "high_limit_check" of OO1 program, compare [fault_selected_value] with [high_limit] - ([high_limit] - [low_limit]) * 0.02.
Error Type	Programer's typing error. <NODAL ERROR>

Error TC	2001.1
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->operator_select-->operator_input_from_file
FP IDX NO	6291
IDX	2001.1
Number of Visits Until Failure	1
Error Description	If there are 6 sensors in total, the operator can only choose number 1 to 6 as his selection. The program should deal with the situation when a negative number is accidentally input by the operator as his selection.
Correction	In "operator_input_from_file" of OO1 program, add "b2=lt.int(o_sel,"0")" to deal with the accident negative input by the operator.
Error Type	Undefined situation by the specification. <NODAL ERROR>

Error TC	2007.2
Fmaps Until Error	svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s-->fault_select-->failed_validation-->last_valid_check1
FP IDX NO	2007.2
IDX	1651.2
Number of Visits Until Failure	5
Error Description	This is a coding error concerning initial value of a helping variable [mindev].
Correction	In "last_valid_check1" of OO1 program, instead of using the initial value "2000" for [mindev], use -1. In "sel_min_dev", if given/input initial value for [mindev] is "-1", use the deviation of the first good sensor as the initial value for [mindev].
Error Type	Coding error. The programming is loose when dealing with initial value. <NODAL ERROR>

Table 7-1 SVA Error Details

7.9. Calculate Node Visiting Frequency

“search_uncovered_simple” under OO1 is built to calculate how often every node of a given function map is visited throughout a set of test cases. The inputs to 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 than or equal to zero. If the register of a node has zero value, it means that 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. Each string ends with the number of visits to this node and the visiting frequency of this node.

[search_uncovered_simple]

Inputs:

FNO: 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:

FNO: /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

[0_499.svatcsidx.fn]

6

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpa
ths_after2000/decpaths0/
0_38²¹

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/dec
40_99

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/dec
100_199

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/dec
200_299

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/dec
300_399

/usr2/oo1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/decpaths_after2000/dec
400_499

Outputs:

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

[0_499.strset.omap]

svaTest_s-->svaGUItest_s-->k(DEC:input_suc_or_not | ALTNO:2)22 TIMES: 12023 FRE:
0.00666778²⁴

svaTest_s-->svaGUItest_s-->DOsvaGUI(DEC:input_suc_or_not | ALTNO:1) TIMES: 17877 FRE:
0.993332

svaTest_s-->svaGUItest_s-->RunMenu_s--

>SVA_process_s(DEC:outof_range_or_not | ALTNO:2) TIMES: 17877 FRE: 0.993332

svaTest_s-->svaGUItest_s-->RunMenu_s-->range_error(DEC:outof_range_or_not | ALTNO:1)
TIMES: 0 FRE: 0²⁵

.....

²¹ "search_uncovered_simple" will try to find a file named "0_38.coveragedb.omap" under directory
".../decpaths0"

²² Path from the function map's top node to this node.

²³ Number of visits to this node.

²⁴ Visiting frequency of this node.

²⁵ This node has not been covered yet.

7.10. 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.

7.10.1. Nodes Needing to Be Tested, But Not Tested

There are 14 untested nodes that should be tested in the 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 that only a small number of nodes can be tested in this way. If we applied “search_uncovered_simple” tool to the 001 SVA program at a much earlier stage and found more leftover nodes that need to be covered, we have to perform more random tests and check out the uncovered nodes at a later stage when the number of nodes that have not been covered is small enough to handle manually. The 14 uncovered nodes that need to be tested are listed below. Each is followed by the logic in the SVA 001 program that leads to that node and the index(es) of the test case(s) that cover that node. Finally, the test 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 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_comparision->k(DEC:type4_or_not | ALTNO:2)

 Under the same situation as in 1, this node is called.

- 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 node is called under the same situation as 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 node is called under the same situation as in 5.
-
- 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.
-
- 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.
-
- 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.
-
- 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.
-

13 svaTest_s-->svaGUltest_s-->RunMenu_s-->SVA_process_s
-->valid_sig_from_psenors_or_not-->calculate_sig_from_psenors
-->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.

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.

7.10.2. Unnecessary Nodes

We found two uncovered nodes in the 001 SVA that are of no use and could not be reached at all. They were deleted from 001 SVA program.

1 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.

2 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.

7.11. 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.

7.11.1. Working Principle of “cut_ iterations”

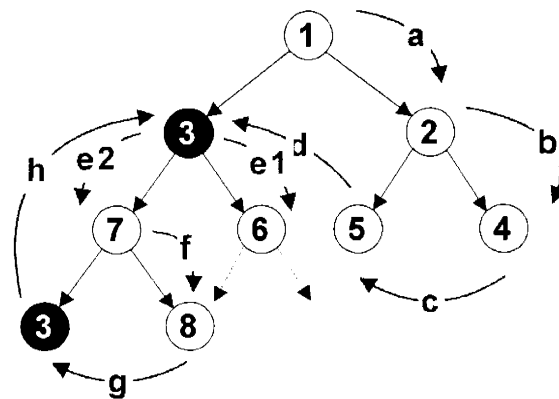


Figure 7-4 A Static Unexpanded 001 FMap Containing Looping Structure

The FMap shown in Figure 7-4 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 from top to bottom and from right to left. If there is no decision node in the program, every node will be executed according to this order.

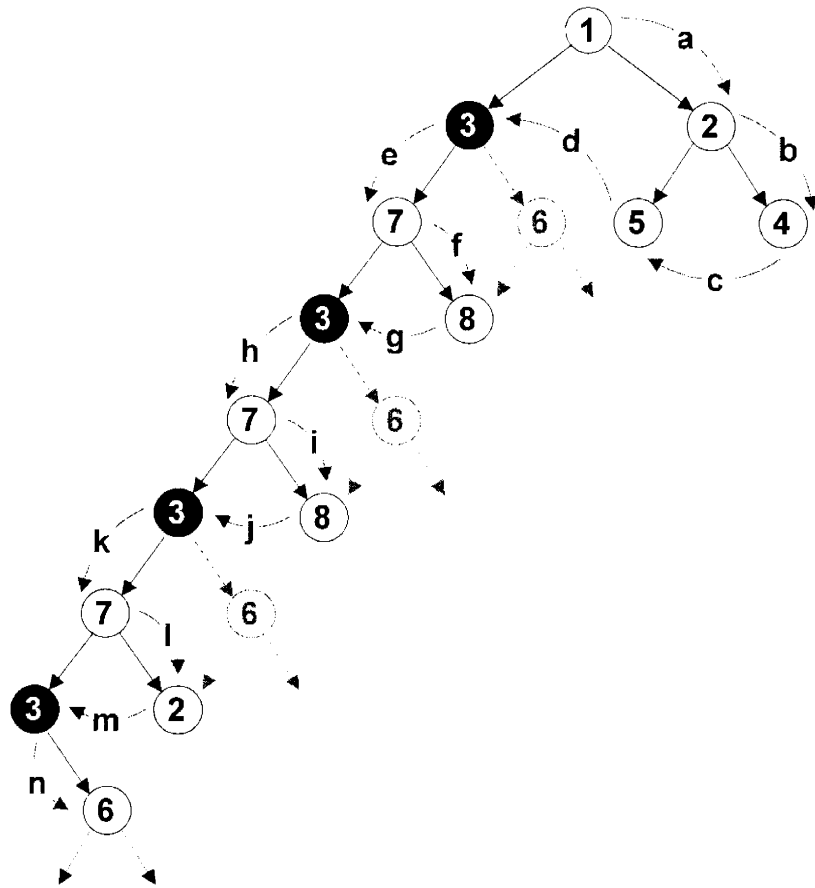


Figure 7-5 A Flowpath with Three Repetitions of a Looping Structure

An execution flowpath of the FMap shown in Figure 7-4 is presented in Figure 7-5. At the decision node, also the loop entrance node, node3, node7 is chosen for three consecutive times, which means the loop is repeated three times in this execution. At the fourth time, the loop exit node, node6 is chosen and the loop is exited. When “cut_ iterations” notices the third repetition of the loop within the given flowpath, it cut all such repetitions beyond the second one. In Figure 7-5, this means that the shaded part is cut from the original flowpath.

7.11.2. Inputs and Outputs of "cut_iterations"

Inputs:

FN: Name of the file that includes names of the flowpath files
"#.#.decpaths.omap" to be scanned by "cut_iterations".
Directory: Directory of FN.

"#.#.decpaths.omap" example:

20:
/usr2/OO1/mt1/sva_project/sva_simple_for_single_input/svatest_s.coveragedb/
0.1.decpaths.omap
0.2.decpaths.omap
.....
0.20.decpaths.omap

Outputs:

Fn.out: Name of the file that includes names of all the output flowpath files
"#.#.decpaths.omap.cut".

"#.#.decpaths.omap.cut" example:

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

7.12. Steps to Identify Unique Flowpaths

7.12.1. 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 case²⁶ 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.

²⁶ 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”.

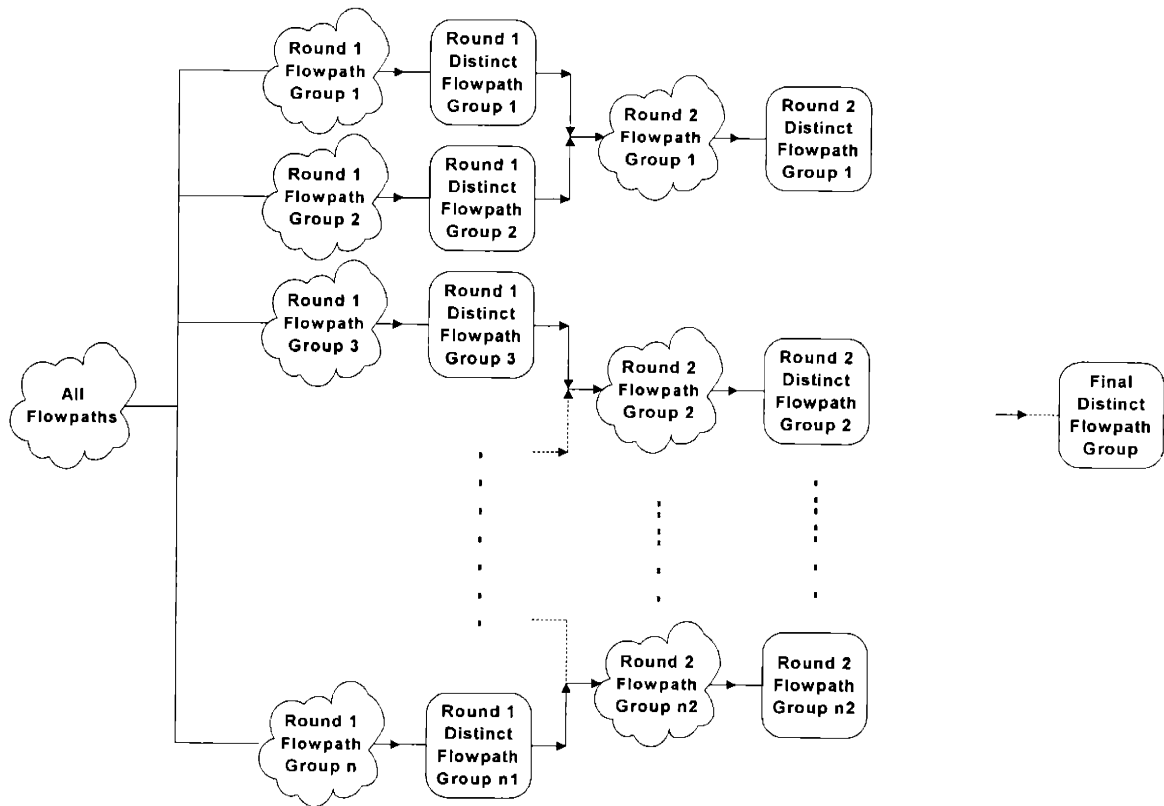


Figure 7-6 Identify Unique Flowpaths Step by Step

7.12.2. 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. The series of commands required to perform in order to achieve this goal is listed as follows.

Under “./decpaths0”

Commands	Output files
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"

Commands	Output files
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"

Commands	Output files
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]

7.13. Count Visiting Frequency of Flowpath

7.13.1. 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 case²⁷ 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

²⁷ 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”.

“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 the total number of distinct flowpaths increases as the total number of test cases increased, “count_diff.pl” can be used.

7.13.2. 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 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 40558 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`

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

The Representative File
of The Flowpath Group

```
[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_99.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
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
175.3.decpaths.omap.cut
.....
```

Files Representing
the Same Flowpath

- `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 1
2 1
3 2
4 3
```

Total Number of Distinct Flowpaths

Total Number of Test Cases

8. References

- Amma01 H. Ammar, T. Nikzadeh and J. Dugan, "Risk Assessment of Software-System Specifications", IEEE Transactions on Reliability, Vol. 50, No. 2, June 2001
- Beiz90 B. Beizer, "Software Testing Techniques", Van Nostrand Reinhold, USA, 1990
- Chen01 M. Chen, M. R. Lyu and W. Wong, "Effect of Coverage on Software Reliability Measurement", IEEE Transaction on Reliability, Vol. 50, No. 2, June 2001
- Cheu80 R. C. Cheung, "A user-oriented software reliability model", IEEE Trans. On Software Engineering, 6(2): 118-125, 1980
- Evan99 W. M. Evanco, "Using a Proportional Hazards Model to Analyze Software Reliability", Software Technology and Engineering Practice, 1999
- Fran88 G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", IEEE Transactions On Software Engineering, Vol. 14, No. 10, 1483-1498 (Oct. 1988)
- Hami92a M. Hamilton, "Increasing Quality and Productivity with a Development Before The Fact Paradigm", Hamilton Technologies Inc. Document, 1992
- Hami92b "Overview: 001 CASE Tool Suite", Hamilton Technologies Inc. Document, 1992

- Hami94 M. Hamilton, "Development Before The Fact in Action", Electronic Design's Special Editorial Supplement: Engineering Software, June 13, 1994, pp22-30
- Haml94 D. Hamlet, "Connecting Test Coverage to Software Dependability", Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on, 6-9, Nov. 1994, pp158-165
- Huan00 C. Huang, S. Kuo and M. R. Lyu, "Effort-Index-Based Software Reliability Growth Models and Performance Assessment", Computer Software and Applications Conference, 2000.
- Jone91 C. Jones, "Applied Software Measurement, Assuring Productivity and Quality", McGraw-Hill, 1991
- Jin01 Z. Jin and J. Offutt, "Deriving Tests From Software Architectures", Software Reliability Engineering, 2001. ISSRE 2001, Proceedings. 12th International Symposium on, 27-30 Nov. 2001
- Kauf99 L. M. Kaufman, J. Dugan and B. W. Johnson, "Using Statistics of the Extremes for Software Reliability Analysis", IEEE Transactions on Reliability, Vol. 48, No. 3, 1999, September
- Kris97 S. Krishnamurthy and A. P. Mathur, "On the Estimation of Reliability of a Software System Using Reliabilities of its Components", Proceedings The Eighth International Symposium On Software Reliability Engineering, 2-5 Nov. 1997
- Kuo01 S. Kuo, C. Huang and M. R. Lyu, "Framework for Modeling Software Reliability, Using Various Testing-Efforts and Fault-Detection Rates", IEEE Transactions on Reliability, Vol. 50, No. 3, September 2001

- Ligg01 P. Liggesmeyer and O. Maeckel, "Quantifying the Reliability of Embedded Systems by Automated Analysis", Dependable Systems and Networks, 2001, Proceedings, The International Conference on, 1-4 July 2001
- Litt92 B. Littlewood and L. Strigini, "The risks of software," Scientific American Now pp. 62~75, 1992
- Litt00 B. Littlewood and L. Strigini, "Software Reliability and Dependability: a Roadmap", the 22nd International Conference on Software Engineering
- Lyu96 M. R. Lyu (Ed.), "Handbook of Software Reliability Engineering", IEEE Computer Society Press, 1996
- Mala94 Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, B. Skibbe, "The relationship Between Test Coverage and Reliability", Software Reliability Engineering, 1994, Proceedings, 5th International Symposium on, 6-9 Nov. 1994
- 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

- Ntaf88 C. Ntafos, "A Comparison of Some Structural Testing Strategies," IEEE Transactions of Software Engineering, Vol. 14 No. 6, 868-873 (June 1988)
- Ouya95 Meng Ouyang and Michael W. Golay, "An Integrated Formal Approach for Developing High Quality Software for Safety-Critical Systems", Massachusetts Institute of Technology, May 1995
- Pham01 L. Pham and H. Pham, "A Bayesian Predictive Software Reliability Model with Pseudo-Failures", IEEE Transactions on Systems man & Cybernetics — Part A: Systems and Humans, Vol. 31, No. 3, May 2001
- Pops01 K. Goseva-Popstojanova, A. P. Mathur, K. S. Trivedi, "Comparison of Architecture-Based Software Reliability Models", Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on, 14-18 May 2001
- Sui98 Y. Sui, "Reliability Improvement and Assessment of Safety Critical Software", Massachusetts Institute of Technology, May 1998
- Tal01 O. Tal, C. McCollin and T. Bendell, "Reliability Demonstration for Safety-Critical Systems", IEEE Transactions on Reliability, Vol. 50, No. 2, June 2001
- Triv01 K. Popstajanova and K. Trivedi, "Architecture based approach to reliability assessment of software systems", Performance Evaluation, Vol. 45, No.2, June 2001
- Vouk94 M. A. Vouk, K. C. Tai, and A. Paradkar, "Empirical Studies of Predicate-Based Software Testing", Software Reliability Engineering, 1994, Proceedings, 5th International Symposium on, 6-9 Nov. 1994

