An Approach for Oracle Data Selection Criterion

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Myung-Hwan Park

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dr. Mats P.E. Heimdahl, Advisor
September, 2010

Acknowledgments

There is an old Korean saying, “All people are born with three lucks in life”.

I undoubtedly devote my first luck to my advisor, Dr. Mats Heimdahl. He has been an extremely proficient and competent guide for a stranger in the academic world. His insightful advice and positive nature has served as the essence of support and confidence for me throughout my Ph.D. He has been extremely patient in guiding me and is a huge source of inspiration. Thank you for everything, Dr. Heimdahl.

The second luck is devoted to my colleague, Matt Staats. Without his help, I could not finish my Ph.D. degree. He has been extremely kind and positive in helping with my experiments and the writing of my thesis. He also made a Lustre translation tool for my experiments and spent his valuable time performing the experiments. My experiments in this dissertation could not have been carried out without Matt’s help and support. Thanks, Matt.

The last luck is dedicated to Dr. JinYoung Choi, my master’s degree advisor in Korea. He has always encouraged me not to settle for the present situation, but to go and see the world. His kind concern and inspiration has been the source of my unwavering committtment during my Ph.D studies.

I would also like to acknowledge the Korea Air Force for allowing me to study in America and supporting me during my Ph.D.

I thank my committee members, Professor Eric Van Wyk, Antonia Zhai, and Donald Kahn for their great service on my thesis. With their generous efforts and time, they helped me accomplish this dissertation.

I would like to thank Dr. YoungDae Kim for his valuable comments on my research and life. Without his help, I would have been extremely lost and confused during my

Ph.D.

I would like to thank my colleagues Michael Whalen, Anjali Joshi, Ajitha Rajan and Kai Xu for their vary useful advice and help in my research. I will never forget my life in Minnesota, enjoying their friendship and professional encouragement.

I would also like to thank Jinoh, Hoonjung, Dongchul and Taehyun for discussing and sharing my concerns about everything during my Ph.D. Without them, my Ph.D courses would be lonely and might have gone off track.

I would like to thank my parents and parents-in-law for always being there for me. Without their support and encouragement , I would never have finished this long Ph.D journey. Special thanks to my parents-in-law for raising my daughter, Sane, during this difficult time of separation from her.

Finally, I would like to thank my lovely wife, HyunE, for motivating, supporting, and standing by me through the years of my Ph.D. A million words are not enough to express my gratitude to her and for her. Thanks forever, HyunE.

Abstract

Testing activities involve the execution of a program under test using input data and the examination of the test results to determine success or failure. One of most important tasks in examining test results is to choose the variables to observe, called *oracle data*, that are used in the examination. Since it is often infeasible to examine all variables of the program under test, we have to choose a subset of the program variables as oracle data. If we can choose variables which can reveal more errors than others, it can significantly increase test effectiveness. A challenge of selecting oracle data is to predict the capability of variables to reveal errors when examined. The work in this dissertation addresses the problem of choosing oracle data to increase test effectiveness.

To predict the error finding capability of variables, we focus on the error propagation behavior of a system. An error in a system can propagate to other variables if those variables are computationally related to the error. Sometimes, however, the error can be masked out during computation. To analyze such error propagation behavior, we propose a novel mechanism of *error propagation analysis* which estimates error propagation behavior through a static analysis of the code. The error propagation analysis computes the error propagation probability for each variable, and this quantitative measure represents the error finding capability of variables.

The second contribution of this work is to establish an oracle data selection criterion. In a naive approach, we may simply pick the variables with the highest error finding capability. This, however, ignored the possibility that several variables may reveal the same error making the selection suboptimal. Our criterion enables us to choose oracle data to increase test effectiveness while the repeated detections of an

error are minimized. The strength of our oracle data selection criterion is its ability to choose oracle data that is effective with a small number of variables in the set of oracle data.

To evaluate the effectiveness of our oracle data selection criterion, we developed an error propagation analysis tool that ranks system variables based on their error finding capabilities. We performed experiments on four sample systems to check the error finding effectiveness of our oracle data selection criterion. The experiment shows promising results in terms of error finding effectiveness. In addition, we investigated the sensitivity of our oracle data selection criterion to changes in the assumptions underlying the approach, and the results indicate that our technique is not very sensitive to even extremely skewed assumptions.

Contents

Acknowledgements . . . . **i**

Abstract . . . . **iii**

List of Tables . . . . **ix**

List of Figures . . . . **xi**

1 Introduction **1**

1.1 Problem Statement 2

1.2 Our Solution 4

1.3 Contributions 6

1.4 Organization of Dissertation 7

2 Background **8**

2.1 Definition of Test Oracle 8

2.2 Techniques of Oracle Data Selection 10

2.3 Error Propagation Analysis 11

2.4 Language Under Consideration 15

2.4.1 An Overview of Our Targeted Language: Lustre 16

2.4.2 Mathematical Model 16

3 Error Propagation Analysis **21**

3.1 Terminology 21

3.2 Assumption 24

3.3 Signal Probability 25

3.3.1 Logical Operator Nodes 26

3.3.2 Relational Operator Nodes 29

3.3.3 *If_then_else* Nodes 30

3.3.4 Signal Probability of a Variable with Feedback Input 31

3.3.5 Algorithm of Computing Signal Probability 33

3.4 Error Propagation Probability 33
3.4.1 Basic Idea 34
3.4.2 Unary Operator Nodes 42
3.4.3 Arithmetic Operator Nodes 44
3.4.4 Relational Operator Nodes 46
3.4.5 *If_then_else* Operator Nodes 48
3.4.6 Logical Operator Nodes 51
3.4.7 Error Propagation Probability of a Variable with a Feedback Input 58
3.4.8 Algorithm of Computing Error Propagation Probability 59

4 Oracle Data Selection **61**
4.1 Definition of Terms 61
4.2 Parameters for the oracle data selection heuristics 62
4.2.1 Signal Probability 62
4.2.2 Fault Distribution 63
4.2.3 Test Case Length 63
4.3 Error Propagation Table 64
4.4 Oracle Data Selection Heuristics: Absolute Ranking vs. Relative Ranking 66
4.4.1 Absolute Ranking 67
4.4.2 Relative Ranking 68
4.5 Oracle Data Selection Heuristics: Interstate Check vs. Last Step Check 70
4.5.1 Interstate Check 70
4.5.2 Last Step Check 71

5 Implementation **73**
5.1 System Overview 73

5.1.1 System Architecture . . . 73
5.1.2 Translator . . . 74
5.1.3 Signal Probability Analyzer . . . 75
5.1.4 Error Propagation Probability Analyzer . . . 76
5.1.5 Fault Model Generator . . . 76
5.2 Implementation Issues . . . 77
5.2.1 Dependence Problem . . . 77
5.2.2 Tree Structure vs. Graph Structure . . . 78

6 Evaluation **80**
6.1 Goal of the Experiments . . . 80
6.2 Case Examples . . . 82
6.2.1 Flight Guidance System . . . 82
6.2.2 Wheel Brake System (WBS) . . . 82
6.2.3 Sensor Voting Model . . . 83
6.2.4 ASW . . . 83
6.3 Evaluation of Effectiveness . . . 84
6.3.1 Experiment Setup . . . 85
6.3.2 Mutant Generation . . . 87
6.3.3 Test Input Generation . . . 88
6.3.4 Baseline Ranking Generation . . . 89
6.3.5 Chi-square Test . . . 90
6.3.6 Experiment Results . . . 92
6.4 Evaluation of Sensitivity . . . 100
6.4.1 Rank Distance . . . 101
6.4.2 Critical Variable Variation . . . 103
6.4.3 Experiment Setup . . . 105
6.4.4 Experiment Results . . . 106

6.5 Discussion 109
6.5.1 Effectiveness of Our Oracle Data Selection Criterion 109
6.5.2 Accuracy of Our Oracle Data Selection Criterion for identifying Critical Variables 111
6.5.3 Sensitivity of Our Oracle Data Selection Criterion 113
6.5.4 Effect of Ranking Variation on Fault Finding Capability of Oracle Data 114
6.6 Threat to Validity 117

7 Conclusion and Future Research Directions **119**

Bibliography **122**

A Algorithm for Relative Ranking **126**

List of Tables

2.1 An example of Lustre program 17

2.2 Semantics of nodes in timed data flow graph 20

3.1 Computation of signal probability for logical operators 26

3.2 Signal probability of relational operators 31

3.3 A simple program 36

3.4 Delayed error propagation 43

3.5 Error propagation time of the example in Table 3.4 43

3.6 Formula to compute error propagation attributes for a unary node . 44

3.7 Computations of EPAs for error independent arithmetic node 45

3.8 Computations of EPAs for an error dependent arithmetic node . . . 45

3.9 Computations of EPAs for error independent relational node 48

3.10 Computations of EPAs for error dependent relational node 49

3.11 Computing EPAs for the error independent *if-then-else* node 50

3.12 Computing EPAs for the error dependent *if-then-else* node (non-boolean node) 51

3.13 Computing EPAs for the error dependent *if-then-else* node(boolean node) 52

3.14 Computation of EPAs for an error independent node 55

3.15 Computation of EPAs for error dependent node (r: fault probability) 57

4.1 Error propagation table for the timed data flow graph in Figure 4.1 . 66

4.2 Failure probability of variables in Table 4.1 68

4.3 Failure probabilities of nodes for three evaluation steps 71

6.1 An Example of a Contingency Table 91
6.2 An example expression of the Sensor Voter system 95
6.3 Hypotheses Evaluation 99
6.4 Base Rank Distance 103
6.5 Sensitivity to signal probability 107
6.6 Sensitivity to Fault Model 108
6.7 The rank of critical variables in baseline ranking 112

List of Figures

2.1 Richardson's and Binder's definition of test oracle. 9

3.1 A time data flow graph with two errors 24
3.2 Computation of signal probability of the AND operator node 26
3.3 AND operator node with dependent predecessor nodes 27
3.4 Computation of signal probability with symbolic representation . . . 29
3.5 The relation between signal probability and number of evaluation . . 32
3.6 Timed data flow graph of expression: $B := A < (4 + 3)$ 36
3.7 Computation of error propagation attributes (EPAs) 37
3.8 An error independent logical node 53
3.9 An error dependent node 55
3.10 A simple error dependent node 58
3.11 A variable of which output is fed back 59

4.1 A timed data flow graph with the predetermined signal probability for input nodes 65

5.1 System overview 74
5.2 A Lustre program and its error propagation trees 75
5.3 Dependence between inputs 78
5.4 The tree and graph structure 79

6.1 Effectiveness of oracle data: baseline ranking vs. random manner . . 94
6.2 Effectiveness of oracle data: relative ranking vs. absolute ranking . . 96
6.3 Fault finding capability of skewed rankings(FGS system) 115

6.4 Fault finding capability of skewed rankings(WBS system) 115
6.5 Fault finding capability of skewed rankings(Sensor Voter system) . . . 116
6.6 Fault finding capability of skewed rankings(ASW system) 116

A.1 Recompute the error propagation probability 127

Chapter 1

Introduction

Testing is one of the most important processes in software development. Testing activities involve the execution of a program under test using input data and the examination of the output to determine success or failure. A fundamental assumption of testing is that there is a mechanism, *a test oracle* (*oracle* for short), that determines whether a system under test has behaved correctly or not. Clearly, a test oracle is one of the most important artifacts of the testing process. Weyuker insisted that a program is non-testable if an oracle does not exist [40]. Bertolino identified a test oracle as an important challenge facing current and future testing research [6].

Unfortunately, although oracles are central to any type of testing or run-time verification activities [22, 15, 23], little attention has been paid to test oracles. When we investigated five testing books commonly used by practitioners [8, 27, 21, 29, 7], surprisingly, only one book [7] contained a chapter on test oracles. Furthermore, the concept of test oracle was not even introduced in the other books.

The lack of interest in test oracles seems to originate from a fundamental assumption — known as the *oracle assumption* [40]— of software testing research; the assumption that an oracle is always available and a tester can always talk about the success and the failure of tests. Although this assumption makes research in testing easier, it is simply not true in practice and the oracle selection problem remains a serious challenge.

There are numerous issues related to test oracles that further research should address. For example, more effort has to be put into understanding what an oracle

really is, what properties an oracle should have, and how the actual output is compared with an expected output. Among these issues related to test oracles, in this dissertation we focus our attention on *oracle data*; the set of system variables that we choose to observe during a test. Since the evaluation of test results is performed based on the value of the oracle data, the selection of oracle data may significantly affect test effectiveness. The goal of this dissertation is to provide guidelines for selection of oracle data for effective testing.

Before going into detail about related work and our proposed research, we first clarify the terms; *faults, errors*, and *failures*. *Faults* are mistakes in the program source code such as faulty instructions or missing data or extra instructions. When a fault is executed and it results in an erroneous value in a system variable, it becomes an *error*. If an error propagates to an output variable, it become a *failure* [18].

1.1 Problem Statement

There is some evidences that oracle data may critically impact the capability of finding errors during testing. Some research [32, 43, 9] showed that changing oracle data while the test input remains the same leads to significant changes in the effectiveness of testing. These evidences raise questions related to what oracle data are the most effective in revealing errors.

In practice, the software's output variables have been considered as the only oracle data and provide the sole evidence whether a program under test behaves correctly or not. This practice is based on the belief that an error in the system, if it exists, will eventually affect the value of output variables. Nevertheless, during testing, there are many cases where errors in a system "vanish" before they reach output variables. For example, under the following conditions errors do not propagate to output variables and cause a false negative if only output variables are used as oracle data.

1. *Time-Delay*: A fault results in an error in the program state, but does not propagate to output variables because the test terminates before the error has had a chance to propagate to an output.

2. *Masking*: An error may be masked out during the computation process. For example, in a condition statement such as *"if (a > 3)"*, 4 is assigned to a, but the correct value of a is 5. This error might not be caught since the error was masked out in the relational expression.

3. *Recovery*: An error may interact with another error and lead the system to recover from the error. For example, in the expression "$x * x$", an incorrect sign error in x will not be detected.

As shown in the examples, an output variable is not always a good candidate to be included in the oracle data. To cope with this disappearance(or masking) of errors, researchers attempt to look at internal variables of the system during execution.

Runtime verification [22, 15, 23] and assertions [38, 35, 37] are approaches used to observe the internal variables during system execution to catch errors without needing them propagate to outputs. In these approaches, testers do not wait passively until errors propagate to output variables. Instead, they actively try to seek out the places where errors may occur and monitor appropriate internal state variables. Unfortunately, there are no guidelines for the decision of which variables to monitor. In practice, monitoring all system variables during execution is expensive and often outright infeasible. Thus, we have to choose a subset of the system variables as oracle data. To the best of our knowledge, there is no previous research that has clearly suggested a guideline for choosing oracle data. More specifically, the following question has not been addressed: *Which variable, if monitored, has the highest probability of revealing errors in a system?* The primary aim of this dissertation is to answer

this question. In other words, the goal of this research is to provide an oracle data selection criterion for the efficient selection of variables that are likely to reveal errors during testing.

1.2 Our Solution

To establish an effective criterion for oracle data selection, we propose a novel mechanism of *error propagation analysis* which estimates error propagation behavior through syntactic analysis of the program code. The key concepts of error propagation analysis are *propagation* and *masking*. Propagation of an error indicates that an erroneous value of a variable contaminates the values of other variables. Masking of an error means that an error is masked out before it propagates to the variables that are monitored. Error propagation analysis exploits the properties of these two key factors in its estimation of error propagation behavior. The following explains the high-level ideas of our approach based on error propagation analysis.

Let variable v be a potential source of an error in a system, and assume that we are interested in examining the propagation behavior of an error in v. In our approach, we track the propagation of the error to other variables in the program. As we follow a particular error propagation path, the probability of the error being propagated to another variable in the path decreases as the distance from the original error location increases.

Given a randomly chosen error location v_i, what would be the chance that the error in v_i will propagate to another variable v_j? We define $EP(v_i)$ as the probability of v_i having an error and $PP(v_i, v_j)$ as the conditional probability that the error in v_i is propagated to v_j, given an error in v_i. Thus, a higher $PP(v_i, v_j)$ value means that the error in v_i is more likely to propagate to v_j (or v_j is likely to be infected by the error in v_i). Conversely, if $PP(v_i, v_j)$ is low, the error is less likely to propagate to

v_j. We derive an oracle data selection criterion by using this property of $PP(v_i, v_j)$.

We define *failure* probability of v_j based on $PP(v_i, v_j)$. Let $v_1, ..., v_n$ be possible error locations in a system and $EP(v_1) + EP(v_2) + ... + EP(v_n) = 1$, then the failure probability of v_j is defined:

$$FP(v_j) = \sum_{i=1}^{n} (EP(v_i) \times PP(v_i, v_j))$$

If $FP(v_j)$ is high, the errors in other variables are likely to propagate to variable v_j. In other words, the variable v_j is likely to collect errors in other variables. On the other hand, when $FP(v_j)$ is low, errors in other variables are not likely to propagate to v_j; thus, variable v_j does not collect as many errors as a variable with a high failure probability does.

In our oracle data selection criterion, we choose variables with a high failure probability as the oracle data. The intuition behind this is that if a variable has a high probability of being infected by errors of other variables, monitoring this variable will be more efficient in finding errors than monitoring other variables with a low probability of infection.

There are some previous researches on oracle data selection based on error flow analysis. Our approach, however, is distinct from previous research in three aspects.

First, to our knowledge, no previous research considers the repeated detection of an error in several oracle data variables when they choose oracle data. From a test effectiveness point of view, the repeated detection of an error does not increase the test effectiveness; one catch of an error is enough to meet the test goal. In our approach, we choose oracle data to increase the test effectiveness by reducing the repeated detection.

Second, previous researches did not provide a quantitative weight to each oracle datum [43, 32, 9]. This may cause a waste of resources. Some oracle data can show

better performance in revealing an error than other oracle data, and we can spend more resources to monitor more important oracle data if this oracle data can be identified. However, previous research provides no guidelines for prioritizing oracle data. In our approach, we rank every variable according to its potential to reveal errors. Thus, we believe we can choose a small number of oracle data that will demonstrate high effectiveness of finding errors.

Third, most of the previous research [39, 11, 16] estimated the error propagation probability in a program through empirical experiments. They first seed an error in a program and run the program with test inputs to check whether the error propagats to one of output variables or not. We measure the error propagation probability with the static analysis of the syntactic structure of a program without executing the program. Since our approach does not depend on test inputs, we believe it will be more precise and consistent. We also believe that our analysis is more scalable than the experimental analysis because it does not need to execute the program with a large number of test cases to find the suitable oracle data.

1.3 Contributions

The key contributions of this dissertation are the following:

- An error propagation analysis based on propagation and masking probability. The underlying intuition of this analysis is adapted from hardware testing [4]. We extend this approach to the software testing domain.

- A technique to rank variables in the program in terms of the effectiveness in revealing errors. This rank information is used for oracle data selection.

- A tool for a thorough evaluation of the validity of our approach. The tool can

perform an error propagation analysis, and rank the system variables in terms of their effectiveness in revealing errors.

- An empirical evaluation of the effectiveness and sensitivity of our oracle data selection criterion. We evaluate the effectiveness and sensitivity of our oracle data selection criterion using four case examples.

1.4 Organization of Dissertation

This dissertation is organized as follows. In chapter 2, we provide background information on test oracles and error propagation analysis. We also define the language under consideration and a mathematical model for our error propagation analysis. Chapter 3 describes our error propagation analysis. We first define the terminology and then present our method of error propagation analysis. We present an oracle data selection criterion based on error propagation analysis in Chapter 4. In Chapter 5, we illustrate the implementation of our tool for the oracle data selection criterion. We also discuss major pragmatic issues encountered during implementation. Chapter 6 describes an evaluation of the effectiveness and sensitivity of the oracle data selection criterion. Finally, Chapter 7 summarizes the work in this dissertation and suggests directions for future work.

Chapter 2

Background

This chapter surveys previous works on the definitions of test oracles and techniques for oracle data selection. We also present related works on the error propagation analysis of software and hardware systems. Finally, we introduce the language and mathematical model we focus on in this dissertation.

2.1 Definition of Test Oracle

The notion test oracle is defined differently by various researchers. Howden [17], who first used the term - "test oracle" - in the test literature, defined it as "a program specification, a table of examples, or the programmer knowledge on how the program should operate". More specifically, a test oracle of a program P is a source of information about a hypothetical correct program of P.

According to Hamlet [14], a test oracle is a predicate J such that $J(x, y)$ holds *iff* : $x \notin dom(S)$ or $(x, y) \in S$ where x is an input, y is an output of a program and S represents the specification, a set of ordered input-output pairs describing allowed behaviors of the program.

Voas [39] also defined a test oracle as a predicate, which is intuitively similar to Halmet definition. When a function g represents a program, and a function f stands for the ideal implementation of g, a test oracle w is a predicate such as $w(x, y)$ is $TRUE$ iff $f(x) = y$

While Howden's definition emphases the prophetic ability of a test oracle, Hamlet's and Vaos's definitions focus on an oracle's ability to evaluate test results. Richardson's [34] definition includes both aspects. She mentions that the definition of a test oracle contains two parts: the *oracle information* that is used as an expected output, and the *oracle procedure* which compares the oracle information with the actual output. The oracle information represents the output from a correct version of P and the oracle procedure is the process to determine whether the output from P is the same as the output from a correct version of P.

Binder's [7] definition of a test oracle is consistent with Richardson's. He states that a test oracle consists of a result generator to produce an expected output for an input as well as a comparator to check the actual output against the expected output. Figure 2.1 depicts the test oracle definition suggested by Richardson and Binder.

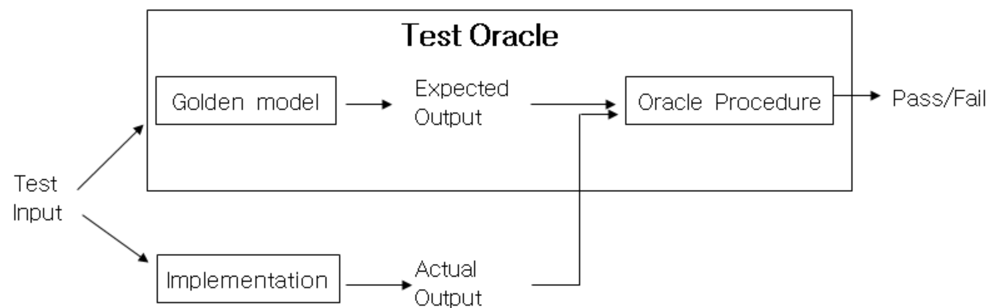


Figure 2.1: Richardson's and Binder's definition of test oracle.

Kaner [20] also insisted that a complete oracle would have three capabilities and carry them out perfectly: a generator to provide the predicted or expected results for each test; a comparator to compare the predicted and obtained results; and an evaluator to determine whether the comparison results are sufficiently close to be a pass.

Machado [24] defined a test oracle by using a specification. In his contexts, the

specification S is represented by a test case, TC, which is defined as a total function from elements of the input domain to elements in the output domain. Thus, $TC(t)$ denotes the corresponding target output for a given input t. The program p meets TC if and only if $p(t) = TC(t)$ for all $t \in Dom(TC)$. A function O is called an oracle for p on TC if for all $t \in D$(input domain)

$$O(t) = \begin{cases} true & \text{if } p(t) = TC(t), \\ false & \text{if } p(t) \neq TC(t), \\ true & \text{if } t \notin dom(TC). \end{cases}$$

2.2 Techniques of Oracle Data Selection

Anderson [2] pointed out that monitoring internal behaviors can quickly reveal errors. Not only can it prevent damage from propagating through the system, but it also enables simplification of error recovery and fault treatment.

Xie and Memon [43] used the state of a GUI at a particular time as oracle data in GUI testing. A GUI is modeled as a set of widgets (*e.g.*, buttons, panels, text fields) that constitute the GUI, a set of properties (*e.g.,* color, size, font) of these widgets, and a set of values (*e.g.,* red, bold). A GUI state is composed of the non-empty set of triples {widget, property, value }. They chose three types of oracle data with an increasing level of details, namely, widget, active window, and all windows. The widget is the least descriptive oracle data that contain a single triple, describing one value of a property of a single widget. All windows data are the most descriptive oracle data that contain values of all properties of all the widget. They ran experiments to check the effectiveness of oracle data to reveal errors. The results showed that the oracle data using all-windows was best, whereas,oracle data using a widget was the worst in test effectiveness.

Rajan *et al.* [32] showed that both monitoring and comparing every system variable is more effective in finding errors than only comparing output variables. The purpose of their experimentation was to check the quality of two different test suites that satisfy the Modified Condition and the Decision Coverage(MCDC) test adequacy criterion. When they ran experiments, they used two different sets of oracle data. One contained only the output variables, while the other contained both the output variables and all internal variables. The results showed that the latter oracle data revealed up to 22% more errors than using only output variables.

Briand *et al.* [9] investigated the test efficiency of the two different types of oracle data: the concrete state and abstract state of an object. The concrete state of an object is the set of values of private and public variables at a particular time. On the other hand, the abstract state of an object consists of the values of public variables. The experimental results demonstrated that oracle data consisting of the concrete state are more effective than those of the abstract state.

From these experiments, it is clear that the choice of oracle data is crucially important for the effectiveness of testing. The challenge is to a priori determine which variables to include in the oracle data.

2.3 Error Propagation Analysis

Goradia [11] proposed a *dynamic impact analysis*, a technique to measure the impact of an error in a variable on other variables during program execution. If an error in variable x is highly likely to cause an error in variable y, the impact strength of x on y is high. In contrast, when the impact strength of x on y is low, y is insensitive to an error in x. The goal of this dynamic impact analysis was to estimate the impact strength from every variable to the output variables. The impact strength is the product between the probability that a specific error occurs and the probability that

the error transfers to an output variable. The computation of the impact strength is conducted through an empirical analysis. For operations such as a data transfer operations, the probability of error transfer is always 1.0 because any error in x is directly reflected as an error in y. For computation operations, the error transfer probability is estimated by the following approach: randomly pick some number of alternate values of x from the error set, perform the operation with each of the alternate values, count the number of times the resulting value of y is erroneous, and compute an estimate for the above probability. A dynamic impact analysis is used to measure the error propagation sensitivity of every variable and operator. However, this technique does not provide any theoretic basis for the computation of the error transfer rate, limiting their approach to their specific environments. This technique also requires a predefined error set for each variable or operator. This requirement is often unobtainable. Finally, performing this empirical analysis for all computations in a program can be prohibitively time consuming.

Voas [39] proposed a *sensitivity analysis* to rank program locations according to their ability to affect the program's computation. The sensitivity of program location l is an estimate of the minimum probability that a fault in l will result in an output error under a specific input distribution. If location l is assigned a sensitivity of 1 under a particular input distribution, it indicates that each input executing l will result in software failure if l contains a fault. If l is assigned a sensitivity of 0, it is anticipated that regardless of what fault occurs in l, no input will cause the fault to propagate to an output. The sensitivity of location l depends on three factors: execution probability, infection probability, and propagation probability. Sensitivity analysis is a very similar approach to dynamic impact analysis since they both deal with the probability that an error is propagated to the output. However, while the propagation probability of sensitivity analysis represents the possibility that an error

in location l propagates to the output for all possible inputs, the impact strength of the dynamic impact analysis, in contrast, refers to the possibility that an error injected in location l propagates to the output for a specific execution.

Abdelmoez *et al.* [1] suggested a technique to estimate the error propagation probability between software components in a software architecture. The error propagation probability, $EP(A, B)$, is the probability that an error in component A is propagated by B because the output of B will change as a result of an error that occurred in A. An error propagation matrix is constructed from the error propagation probability among the components. If a component has a larger value in one matrix column, it is likely to be affected by errors in other components. Monitoring this component during test execution can reveal more errors since errors in other components are likely to transfer to this component. The estimation of error propagation between components is based on an analytical method that uses the state information of components and set of messages passed among components. The challenge of this approach is, however, to obtain the precise state information and probability of a message to pass from a component to the other component. For larger systems, these information is almost unobtainable, and thus the availability of this approach may be limited to a small system.

Hiller *et al.* [16] proposed a framework-Exposure, Permeability, Impact and Criticality (EPIC)- aimed at providing a means for profiling software such that weakness(a module to which errors cannot propagate) and hot-spot(a module to which most errors originated from other modules can propagate) in modular software can be identified. Using this approach, they introduced the measure of error permeability that indicates the propagation degree of an error occurring on the input of a software module to outputs of the module. The error permeability is the conditional probability of an error occurring on the output given that there is an error on the input. The measure

of the error permeability of modules is based on experimental estimation. That is, they inject faults into a system and monitor the execution trace of the system. If a module has a high probability of error permeability, it may be more cost effective in catching errors than any other modules with a low probability of error permeability. However, to obtain the error permeability of modules, an experimental analysis has to be performed, and the analysis will take a significant time for a large system. Thus, the availability and consistency of the experimental analysis are also limited.

Asadi and Tahoori[4] presented an approach to compute the error propagation probability for the estimation of system failure rates due to soft errors at the logic levels. To compute the error propagation probability, this approach first assumes an error in a gate. Then, it extracts the structural paths from the error gate to all reachable gates and traverses these paths to compute the propagation probability of the error in the gate. The approach distinguishes the *on-path* signal and *off-path* signal. The *on-path* signal is a signal on the path from the error gate to a reachable output gate. The *off-path* signal is a signal that is not an *on-path* signal. For an *off-path* signal, the approach uses the signal probability [28], which is the probability that the signal has value 1. The signal probability is used to compute the masking probability for each gate in a circuit. The masking probability is the probability that the errors in the input of a gate are masked out during the operation of the gate. The masking probability of a gate primarily depends on the signal probability of the inputs of the gate. We will discuss the signal probability in Chapter 4 in detail. For an *on-path* signal, the approach uses four different probabilities:$P_a, P_{\bar{a}}, P_1$, and P_0. These probabilities will be explained in detail in Section 3.4.1. Asadi's work is the foundation of our error propagation analysis and will be also discussed in Section 3.4.1. We adopt their idea of *on-path* signal probabilities and formulas for AND, OR and NOT gates. However, their approach is restricted to error propagation in

the hardware circuit domain. We develop several additional formulas and analysis techniques to extend their approach to the software domain. Our analysis is also different from Asadi's work in that our analysis can handle dependancies among inputs to an operator. We discuss the dependence problem in Chapter 4 in detail.

2.4 Language Under Consideration

In this dissertation, we focus our interest on the synchronous data flow language[13], which is now in use in many embedded software(avionics, nuclear power plant, medical device, etc). This language is based on the synchronous paradigm. In this paradigm, the behavior of a system is a sequence of reactions, each reaction is to read current inputs, update internal states and compute outputs. The states are the valuation of the memory(variables) and the reaction is *instant*, meaning that it take a no time. The instant reaction in the synchronous paradigm makes the behavior of the system to be fully deterministic from the function and time point of view. Examples of the synchronous language are Esterel [5], Argos[12], Lustre [10] and Signal [12].

The synchronous data flow languages have several attracting characteristics for our research. First, these languages do not have the concept of iteration and recursion, which otherwise may make our analysis considerably more complicated. Second, the propagation of an error from an error source location to other variables can be well modeled in these languages since the notion of data flow can be customized with the propagation of errors in the system. Finally, these languages generally do not have any side effects on computations; thus, it always makes the implementation with these languages deterministic. This characteristic can also make our error propagation analysis deterministic and relatively simple. Though our error propagation analysis primarily focuses on the domain specific language, we believe this analysis can be applied to general programming languages.

2.4.1 An Overview of Our Targeted Language: Lustre

In this section, we briefly introduce the Lustre language, which is the basic programming language under consideration in our error propagation analysis. This description is referred from [13]. A Lustre program operates on flows of values. Any variable X represents an infinite sequence of values such as $(x_0, x_1, ..., x_n, ...)$. The x_n is the value of X at the nth time step. Output flows are defined by an equation. For example, "$X = E$" represents that the value of X is same with the value of E at the same time step. Lustre provides two temporal operators.

- "pre": the pre operator provides a method to access the previous value of its argument variable. For example, "pre(X)" represents a flow starting by *nil* such as $(nil, x_0, x_1, ...)$.
- "$\rightarrow$": This operator is used to define initial value. For example, "$X \rightarrow Y$" is the flow $(x_0, y_1, ..., y_n, ...)$, starting the initial value with X and then equal to Y.

Table 2.1 shows a simple example of a Lustre program. The program represents a counter of events. It takes two boolean inputs("evt" and "reset") and returns the number of occurrences of event which occurred since the last "reset".

2.4.2 Mathematical Model

This section defines the timed data flow graph to be used as a basic model for our error propagation analysis. The semantics of nodes in the timed data flow graph is defined in Table 2.2.

Definition 1 ***Timed data flow graph:***
A ***timed data flow graph*** *G is a tuple $G = (V, E, C)$ where*

1.	$Node\ Count(evt,\ reset:\ bool)\ returns(count:\ int);$
2.	let
3.	$count\ =\ if(true\ \rightarrow\ reset)\ then\ 0$
4.	$else\ if\ evt\ then\ pre(count)\ +\ 1$
5.	$else\ pre(count);$
6.	tel

Table 2.1: An example of Lustre program

1. *V is a finite set of nodes.*

2. *$E \subseteq V \times V$ is the set of directed edges. An edge $(n_1, n_2) \in E$ represents a data flow from node n_1 to n_2.*

3. *C is a clock. Time sequence of C, $t = t_1t_2...$, is an infinite sequence of time values $t_i \in N$ with $t_1 = 1$ and $t_{i+1} = t_i + 1$.*

The nodes of the timed data flow graph are expressions and the expressions are evaluated every time step of clock C. The value of a node can be expressed by a sequence of data values $< v_1, v_2, ... >$ where v_i is the value of the node at time i. The edges represent a flow of the value of a node to another node. Clock C is initialized to 1. As time advances, the value of clock C increases 1 per one time step. The value of a node at a specific time can be referred to by function X. Function X is a mapping $X : V \times N \rightarrow D$ where D is a set of domains for data values.

Definition 2 ***Incoming degree, Outgoing degree:***
*The **incoming degree** of a node is the number of incoming edges of the node. The **outgoing degree** of a node is the number of outgoing edges of the node.*

Definition 3 ***Input node, Output node:***

The ***input nodes*** *are nodes with an incoming degree of* 0*. The* ***output nodes*** *are nodes with an outgoing degree of* 0*.*

Definition 4 ***Path:***

A ***path*** *in the graph is a finite or infinite sequence of nodes,* $< n_0, n_1, ..., n_i >$*, where* $(n_j, n_{j+1}) \in E$

Definition 5 ***Reachable:***

Node k *is* ***reachable*** *from node* h *if there exists a path,* $< h, n_i, ..., n_j, k >$*.*

Definition 6 ***Ancestor node:***

Node k *is an* ***ancestor node*** *of node* h *if* k *is reachable from* h*.* $ANCESTOR(h)$ *is the set of nodes that are ancestor nodes of* h*.*

For example, in the path $< h, n_i, ..., n_j, k >$, $ANCESTOR(h)$ is a set of nodes, $\{n_i, ..., n_j, k\}$.

Definition 7 ***Descendant node:***

Node h *is a* ***descendant node*** *of node* k *if* k *is reachable from* h*.* $DESCENDANT(k)$ *is the set of nodes that are descendant nodes of* k*.*

For example, in the path $< h, n_i, ..., n_j, k >$, $DESCENDANT(k)$ is a set of nodes, $\{h, n_i, ..., n_j\}$.

Definition 8 ***Predecessor node:***

Let (n_1, n_2) *be an edge in* E*. Node* n_1 *is a* ***predecessor*** *node of* n_2*. If node* n_i *has a predecessor node, the predecessor node is denoted by* $PRED(n_i)$*. If node* n_i *has two predecessor nodes, the predecessor nodes are denoted by* $LPRED(n_i)$*,*$RPRED(n_i)$*. If node* n_i *has three predecessor nodes, the nodes are denoted by* $C_{PRE}(n_i)$*,* $E_{1PRE}(n_i)$*,* $E_{2PRE}(n_i)$*.*

Definition 9 ***Successor node:***

Let (n_1, n_2) *be an edge in* E*. Node* n_2 *is a* ***successor*** *node of* n_1*. Successor nodes of* n_i *are denoted by* $SUCC(n_i)$*.*

So far, we defined basic terminology in the timed data flow graph. We then explained the semantics of nodes in the graph in Table 2.2. Note in the table that the function X generates a value of input nodes for every time step: $X(n,t) = v \in Domain(n)$. This represents that an input is provided to the input variable for every time step.

Semantics of nodes in timed data flow graph (n = node, t = the value of clock C)		
Incoming Degree	**Node Type**	**Semantics**
0	Input node	$X(n,t) = v \in Domain(n)$
	Value node	$X(n,t) = v$, where v is the value of node n.
1	variable	$X(n,t) = X(PRED(n),t)$
	:=	$X(n,t) = X(PRED(n),t)$
	$Unary_minus$	$-X(PRED(n),t)$
	Not	$Not\ X(PRED(n),t)$
	Pre	$X(PRED(n),t-1)$
2	AND	$X(n,t) = X(LPRED(n),t)\ AND\ X(RPRED(n),t)$
	OR	$X(n,t) = X(LPRED(n),t)\ OR\ X(RPRED(n),t)$
	XOR	$X(n,t) = X(LPRED(n),t)\ XOR\ X(RPRED(n),t)$
	$=$	$X(n,t) = X(LPRED(n),t) = X(RPRED(n),t)$
	$\neq$	$X(n,t) = X(LPRED(n),t) \neq X(RPRED(n),t)$
	$<$	$X(n,t) = X(LPRED(n),t) < X(RPRED(n),t)$
	$>$	$X(n,t) = X(LPRED(n),t) > X(RPRED(n),t)$
	$\leq$	$X(n,t) = X(LPRED(n),t) \leq X(RPRED(n),t)$
	$\geq$	$X(n,t) = X(LPRED(n),t) \geq X(RPRED(n),t)$
	$+$	$X(n,t) = X(LPRED(n),t) + X(RPRED(n),t)$
	$-$	$X(n,t) = X(LPRED(n),t) - X(RPRED(n),t)$
	$*$	$X(n,t) = X(LPRED(n),t) * X(RPRED(n),t)$
	$/$	$X(n,t) = X(LPRED(n),t)/X(RPRED(n),t)$
	$\rightarrow$	$X(n,t) = X(LPRED(n),t)$ if $t = 1$ $X(n,t) = X(RPRED(n),t)$ if $t > 1$
3	If_then_else	$X(n,t) = X(E_{1PRE}(n),t)$ if $X(C_{PRE}(n),t) = true$ $X(n,t) = X(E_{2PRE}(n),t)$ if $X(C_{PRE}(n),t) = false$

Table 2.2: Semantics of nodes in timed data flow graph

Chapter 3

Error Propagation Analysis

Error propagation analysis is a process to estimate the error propagation behavior of a program through static analysis of the program code. In this chapter, we introduce our approach for error propagation analysis that used for an oracle data selection criterion in Chapter 4.

3.1 Terminology

Let a timed data flow graph G' represent a correct program under test, and G represent a faulty implementation of the correct program.

Definition 10 ***Fault:***
A ***fault*** *is the substitution of a node by a different node.*

Definition 11 ***Fault probability:***
Fault probability*,* $FA(n_1, n_2)$*, is the probability that a node* n_1 *is substituted by node* n_2*.* $FA(n_1)$ *is the probability that a node* n_1 *is substituted by a different node.*

Recall that faults were informally defined in the introduction as mistakes in the program source code such as "faulty instructions," "missing data," or "extra instructions." In the timed data flow graph, we use a somewhat restricted notion of a fault. The fault is defined as a replacement of a node by a different node in the timed data flow graph. For example, suppose a correct expression is "$A = B + C$". A faulty

expression of this is "$A = B + D$" where the node C is replaced by the node D. Since our error propagation analysis is based on computations considering one fault in a system, this restricted notion of the faut will be suitable for our analysis. The details and rational for this one fault approach will be discussed in Section 3.2.

Definition 12 ***Error:***
Let n be a node from G, and n' be the corresponding node from G'. An ***error*** *has occurs in node n when $X(n, t) \neq X(n', t)$.*

Definition 13 ***Error probability:***
Let n be a node from G and n' be the corresponding node from G'. The ***error probability****, or $EP(n)$, is the probability that node n has an error, i.e., $EP(n) = Pr(X(n, t) \neq X(n', t))$.*

By this definition, a fault is recognized as an error when the fault causes a node to contain any erroneous value. As an example, statement "$x = 0 * v_1$", a fault with "$0 * v_2$" instead of "$0 * v_1$" does not become an error because the evaluated value of x will always be zero. However, if the statement is "$x = 1 * v_1$", a fault with v_2 instead of v_1 can lead to an error based on the values of v_1 and v_2.

Definition 14 ***Masking:***
Let n be a node from G, and n' be the corresponding node from G'. The error in n is ***masked*** *in $m \in ANCESTOR(n)$ if $X(n, t) \neq X(n', t)$ and $X(m, t) = X(m', t)$ where m' in G' is the corresponding node of m.*

Definition 15 ***Masking probability:***
Let n be a node from G and m be an ancestor node of n. Let n', m' be the corresponding nodes from G'. ***Masking probability****, or $MP(n, m)$, is the probability that an error in n is masked out in node m. Formally, $MP(n, m)$ is the conditional probability that*

m will not have an error, given an error in n , $Pr(X(m,t) = X(m',t) \mid X(n,t) \neq X(n',t))$.

An error might be masked out by the operator node during computation. For example, in expression $'A\ AND\ False'$, any erroneous value of A will not be propagated to the AND node. In this case, an error in A is masked. Operators in our language have their own masking characteristics. That is, some operators have relatively low masking effects, while other operators have greater effects.

Definition 16 ***Propagation:***

Let n be a node from G, and n' be the corresponding node from G'. The error in n is ***propagated*** *to $m \in ANCESTOR(n)$ if $X(n,t) \neq X(n',t)$ and $X(m,t) \neq X(m',t)$ where m' in G' is the corresponding node of m.*

Definition 17 ***Propagation probability:***

Let n be nodes from G and m be an ancestor node of n. n', m' be corresponding nodes from G'. ***Propagation probability****, or $PP(n,m)$, is the probability that an error in n is propagated to node m. Formally, $PP(n,m)$ is the conditional probability that m will have an error, given an error in n, $Pr(X(m,t) \neq X(m',t) \mid X(n,t) \neq X(n',t))$*

The propagation probability of a node is inversely proportional to the masking probability of that node, i.e., $PP(n,m) = 1 - MP(n,m)$. If a node has a great masking probability, the probability that an error propagates through the node will be low, and vice versa.

Definition 18 ***Failure:***

An error becomes a ***failure*** *when the error is monitored.*

Definition 19 ***Failure Probability:***

Let n be a node from G. The ***failure probability****, or $FP(n)$, is the conditional probability that node n will have an erroneous value when n is monitored.*

The observability can distinguish a failure from an error. In other words, a failure is an observable error.

3.2 Assumption

In our error propagation analysis, we need an assumption for the number of faults in the timed data flow graph as follows:

Assumption: There exists only one fault node in the timed data flow graph.

This assumption is needed to solve a problem we encounter during our analysis: a **recovery problem**.

If a graph has multiple faults, there is a probability that the errors generated from the faults can be recovered by the interaction of the errors. For example, in Figure 3.1, suppose node A has an erroneous value 3 and B has an erroneous value 1. The correct value of both nodes A and B should be all 2. If C node is $'+'$ operator, no error will propagate to the value of C since the errors were recovered from each other. In our analysis, it is almost infeasible to compute the probability of errors to be recovered by other errors. By assuming a single fault in the graph, we can eliminate this problem.

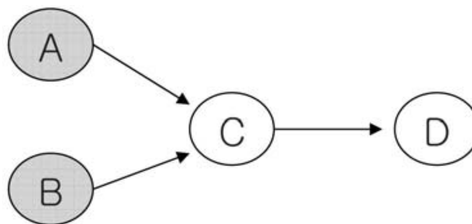


Figure 3.1: A time data flow graph with two errors

This assumption, however, may sound a bit strong for a real system testing sit-

uation where multiple faults can exist. Nonetheless, our analysis based on this assumption will still provide meaningful information for our oracle selection criterion. Suppose a specific node is identified as the most effective node to reveal errors by our oracle data selection criterion under the single fault assumption. It is expected that the node will still show the best (or at least superior) performance to reveal faults even in the case where multiple faults exist since adding more faults in the graph does not deteriorate the fault revealing ability of that node. In this work, we perform our error propagation analysis with the single fault assumption for the above reason, but we consider exploring the non-single fault case as one of our future research issues.

3.3 Signal Probability

Signal probability [4] is the probability that a node will be evaluated to be "true". This probability is used to compute the masking probability of a logical operator node in our error propagation analysis. We first formally define the signal probability as follows.

Definition 20 ***Signal probability:***
Signal probability *of node* n*, or* $SP(n)$*, is the probability that the value of node* n *is true, i.e.* $SP(n) = Pr(X(n,t) = true)$*.*

Since the signal probability is the probability of having value *true* in a node, the computation of signal probability can only be applied to logical operators, relational operators and an If_then_else operator in our language. Signal probability is used to compute error propagation probability, as will be discussed in section 3.4. We begin with logical operators to compute signal probability.

3.3.1 Logical Operator Nodes

The signal probability of a logical operator node is determined by the signal probabilities of the input nodes from which the operator node is reachable. In Figure 3.2, the signal probability of an AND operator node n is computed by the product of the signal probability of $LPRED(n)$ and $RPRED(n)$. This means that evaluation of node n becomes $true$ when the values of both $LPRED(n)$ and $RPRED(n)$ are $true$. The signal probability of OR and XOR operator nodes are computed in a similar way as the AND operator node does. Table 3.1 shows the formula to compute the signal probability for the logical operator nodes.

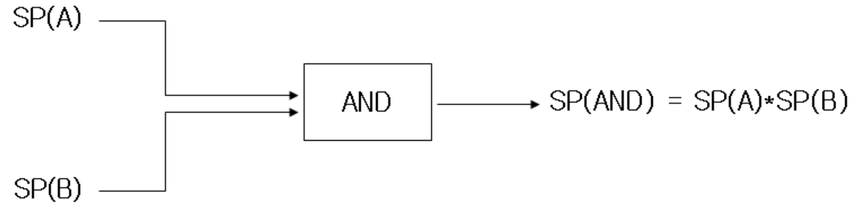


Figure 3.2: Computation of signal probability of the AND operator node

Operator	Signal probability(n= a logical operator node)
AND	$SP(n) = SP(LPRED(n))SP(RPRED(n))$
OR	$SP(n) = SP(LPRED(n)) + SP(RPRED(n)) - SP(LPRED(n))SP(RPRED(n))$
XOR	$SP(n) = (1 - SP(LPRED(n)))SP(RPRED(n)) + SP(LPRED(n))(1 - SP(RPRED(n)))$

Table 3.1: Computation of signal probability for logical operators

When we use the formula in Table 3.1, we have to pay attention to the case where the signal probabilities of predecessor nodes are *dependent* of each other. If two nodes

are dependent, the evaluation of the signal probability of a node is affected by the signal probability of another node. This dependence problem between two nodes happens when both nodes are reachable from a common input node. For example, the AND operator node in Figure 3.3 has two predecessor nodes that are dependent of each other since both nodes are reachable from an input node. If two predecessor nodes $LPRED(n)$ and $RPRED(n)$ of a logical operator node n are dependent of each other, signal probabilities of n can be computed by the following conditional probabilities:

$$SP(n) = Pr(X(LPRED(n), t) = true \mid X(RPRED(n), t) = true)$$

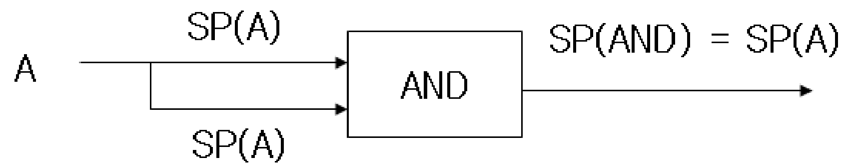


Figure 3.3: AND operator node with dependent predecessor nodes

If a logical operator node has independent predecessor nodes, the computation of the signal probability of that node is straightforward since we can directly follow the formula in Table 3.1. For example, suppose the predecessors of the AND operator node in Figure 3.3 are not dependent and the signal probability is 0.5 for each. Then, the signal probability of the AND node becomes 0.25 (by computing it with the formulas in Table 3.1). On the other hand, if the predecessors are dependent, the signal probability of the AND node becomes 0.5 by the following computation:

$$
\begin{aligned}
SP(B) &= Pr(X(A,t) = true) * Pr(X(A,t) = true \mid X(A,t) = true) \\
&= Pr(X(A,t) = true) * 1 \\
&= Pr(X(A,t) = true) \\
&= SP(A) \\
&= 0.5
\end{aligned}
$$

The key point here is that if the predecessor nodes of node n are dependent of each other, we have to determine common input nodes from which both predecessors are reachable. After that, we can compute the conditional probability of n based on the signal probability of the common input nodes. This determination of common input nodes is not a trivial problem in the graph. One previous study [28] defined a technique for computing the signal probability for logical operators with consideration of the dependent nodes. The basic idea of the study is to assign a character (e.g., input name) to represent the signal probability rather than a number, and then compute the signal probability based on the formulas defined in Table 3.1. This produces a symbolic representation (rather than a number) for the signal probability of the node. Figure 3.4 illustrates the computation process with characters in obtaining signal probability. As can be seen in the figure, every node has a symbol representation for the signal probability.

The next step in the technique is to simplify the symbolic representation. In this simplification phase, any duplicate characters in the representation are removed by a theorem [28] as follows.

Theorem 1 *The product of the signal probability of the same node is equal to the single signal probability of that node.*

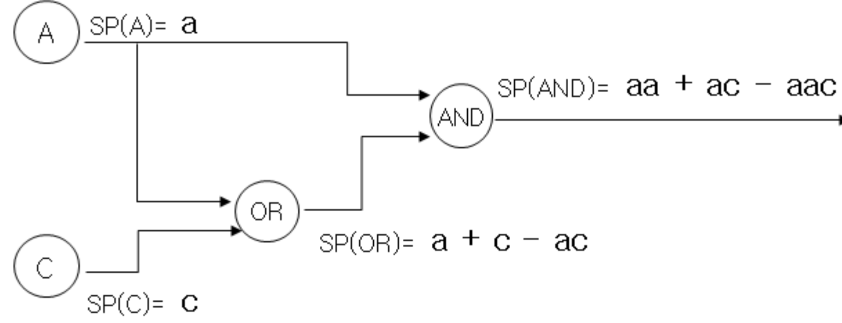


Figure 3.4: Computation of signal probability with symbolic representation

This theorem says that exponents in the symbolic representation for signal probability can be suppressed. For example, a^2 can be suppressed to a. We omit the proof of this theorem. The interested reader can refer to the original paper [28].

By Theorem 1, the symbolic representation "$SP(AND) = aa + ac - aac$" in Figure 3.4 can be simplified as "$SP(AND) = a + ac - ac = a$." The final step is to replace the characters with the actual signal probabilities.

3.3.2 Relational Operator Nodes

Since the domain of relational operator nodes is boolean, we also need to compute the signal probability for the relational operator nodes. The underlying idea to compute the signal probability of the relational operator node is identical to computing the signal probability of the logical operator nodes. The signal probability of the predecessor nodes is used to compute the signal probability of the relational operator nodes. When the domain of the predecessor node of the relational node n is boolean, the signal probability of n can be computed by the formula in Table 3.2. For the formula, we assumed that both predecessor nodes of a relational node n can not be *value* nodes at the same time. For example, we do not consider an expression like $true = true$ in the program under consideration. We also assumed in the formula

that the value node is always located in $RPRED(n)$. If the value node is located in $LPRED(n)$, the signal probability of n is computed in a similar way. One difference of relational operator nodes from the logical operator nodes in computing the signal probability is that the domain of the predecessor nodes of the relational operator nodes is not necessarily boolean. If the domain of the predecessor node of a relational operator node n is not boolean, the signal probability of n does not depends on the signal probability of the predecessor nodes. Rather, the signal probability of n is directly computed from the probability of n to be evaluated as $true$. In the case that n is $>, <, \geq$, or $\leq$, we assign 0.5 to the signal probability of n. Without any knowledge for the value of the predecessor nodes, we believe that 0.5 is the most reasonable probability of n to be $true$. If n is $=$, we assign 0 to the signal probability of n. This is because if the domain of the predecessor nodes of n is non-boolean, the probability of two predecessor nodes holding the same values is approximately 0. For the same reason, we assign 1 to the signal probability of n if n is $\neq$. These formulas are also shown in Table 3.2.

3.3.3 If_then_else Nodes

The domain of If_then_else node n can be boolean when the domain of both $E_{1PRE}(n)$ and $E_{2PRE}(n)$ are boolean. The signal probability of n depends on the signal probability of the predecessor nodes (i.e., $C_{PRE}(n), E_{1PRE}(n)$, and $E_{2PRE}(n)$). The signal probability of n is computed by following formula:

$$SP(n) = SP(C_{PRE}(n)) * SP(E_{1PRE}(n)) + (1 - SP(C_{PRE}(n))) * SP(E_{2PRE}(n))$$

According to the formula, node n evaluates $true$ if (1) both $C_{PRE}(n)$ and $E_{1PRE}(n)$ evaluate $true$, or (2) $C_{PRE}(n)$ evaluates $false$, while $E_{2PRE}(n)$ evaluates $true$.

<table>
<tr><th colspan="3">Signal Probability(n = a relational operator node)</th></tr>
<tr><th>Operator</th><th colspan="2">If predecessors of n are boolean domain</th></tr>
<tr><td rowspan="3">$=$</td><td>$SP(n) = SP(LPRED(n))SP(RPRED(n))$
$+(1 - SP(LPRED(n)))(1 - SP(RPRED(n)))$</td><td>$LPRED(n)$ and $RPRED(n)$
are variables</td></tr>
<tr><td>$SP(n) = SP(LPRED(n))$</td><td>$X(RPRED(n), t) =' true'$</td></tr>
<tr><td>$SP(n) = 1 - SP(LPRED(n))$</td><td>$X(RPRED(n), t) =' false'$</td></tr>
<tr><td rowspan="3">$\neq$</td><td>$SP(n) = SP(LPRED(n))(1 - SP(RPRED(n)))$
$+(1 - SP(LPRED(n)))(SP(RPRED(n)))$</td><td>$LPRED(n)$ and $RPRED(n)$
are variables</td></tr>
<tr><td>$SP(n) = 1 - SP(LPRED(n))$</td><td>$X(RPRED(n), t) =' true'$</td></tr>
<tr><td>$SP(n) = SP(LPRED(n))$</td><td>$X(RPRED(n), t) =' false'$</td></tr>
<tr><th>Operator</th><th colspan="2">If predecessors of n are non-boolean domain</th></tr>
<tr><td>$<, \leq, >, \geq$</td><td colspan="2">$SP(n) = 0.5$</td></tr>
<tr><td>$=$</td><td colspan="2">$SP(n) = 0$</td></tr>
<tr><td>$\neq$</td><td colspan="2">$SP(n) = 1$</td></tr>
</table>

Table 3.2: Signal probability of relational operators

3.3.4 Signal Probability of a Variable with Feedback Input

As explained in 2.4, the timed data flow graph has a clock and every node in the graph evaluates its expression at every time step. The PRE operator is used to refer to the one-step previous value of a node. Thus, the signal probability of the PRE operator node is the same as the signal probability of the node to which it is applied. With the PRE operator node, a node can feed its one-step previous value back to its current input. We first define the feedback input formally.

Definition 21 ***Feedback Input:***
*Let n be a node in the timed data flow graph. If there exists a path, $< n, ..., n >$, the node has **feedback input**.*

If a node has feedback input, the signal probability of the node is affected by the previous signal probability of itself. As a result, the signal probability of the node may change at every time step. Hence, when we compute the signal probability of a node with feedback input, we have to consider how many times the evaluation of the node is repeated. The *number of the evaluation* can be counted by the time passage of the system clock starting from 1.

In our analysis, the number of evaluations is used to compute the signal probability of a node in a specific time step. If a node does not have feedback input, the signal probability of the node dose not change by the number of evaluation. On the other hand, if a node has feedback input, the signal probability changes at every time step. Figure 3.5 shows the signal probability pattern for the AND and OR operator nodes with a feedback input. In the figure, the signal probability of the AND and OR operator nodes in time step 1 is 1 and 0, respectively. The signal probability of the AND operator node converges on 0 as the number of evaluations increase while the signal probability of the OR operator node converges on 1.

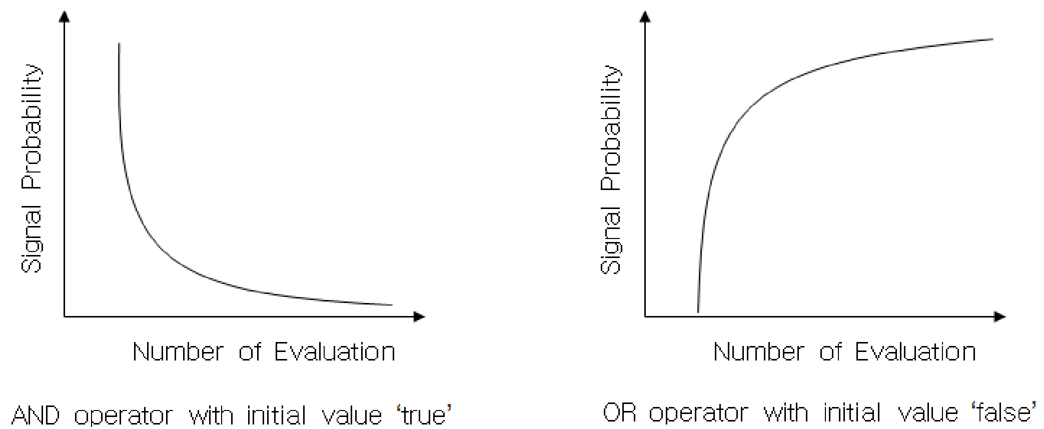


Figure 3.5: The relation between signal probability and number of evaluation

3.3.5 Algorithm of Computing Signal Probability

Algorithm 1 illustrates the computation process of the signal probability for boolean domain nodes.

Algorithm 1 Computation of Signal Probability

1: **for** each boolean domain node n in the timed data flow graph G **do**
2:     **if** n is the input node **then**
3:         Assign a unique id representing the signal probability for n.
4:     **else**
5:         Generate the symbolic representation.
6:     **end if**
7: **end for**
8: **for** each symbolic representation p **do**
9:     Simplify p by Theorem 1.
10:     Replace ids in p with numbers and evaluate
11: **end for**

3.4 Error Propagation Probability

In this section, we explain how to compute the error propagation probability, which depends on the signal probability, the number of evaluations, and operator types. We begin by defining some terms used in this section.

Definition 22 ***Error source node:***
An ***error source node*** *is a node in the timed data flow graph where a fault occurs and its value becomes different from the value of the corresponding node in the correct timed flow graph.*

Definition 23 ***Error dependent node:***

Error dependent node, *n is a node where more than one path exists from the error source node to n.*

Definition 24 ***Error independent node:***

Error independent node, *n is a node where only one path exists from the error source node to n*

3.4.1 Basic Idea

We first introduce Error Propagation Attributes (EPAs), and then give a simple example to show how to employ EPAs to compute error propagation probability.

Error Propagation Attributes (EPA)

EPAs are essential for computing the error propagation probability. The underlying idea of these attributes is adopted from the on-path signal probabilities proposed by Asadi et al. [4]. In this work, EPAs are used to carry the masking and propagation probabilities of the error to all the nodes reachable from the error source node. We use 6 attributes: 4 attributes ($P_a, P_{\bar{a}}, P_1$, and P_0) for boolean nodes and 2 attributes (P_p and P_m) for non-boolean nodes. Following are detailed explanations for the attributes.

- Boolean Domain Attributes

    1. $P_a(n)$ is defined as the probability that node n has an error and the value of n is the same as the value of the error source node. For example, if the error source node has value $true$, $P_a(n)$ is the probability that the value of n is $true$ but the correct value of the node should be $false$.

If the error source node does not belong to the boolean domain, $P_a(n)$ is defined as the probability that node n has an error and the value of n is the same as the value of the first reachable boolean node from the error source node. Figure 3.6 is an example of this concept. Suppose the node with value 4 in the figure is the error source node. The domain of the error source node is not boolean and the first reachable boolean node is B. Thus, computing P_a should refer to the value of B.

2. $P_{\bar{a}}(n)$ is defined as the probability that node n has an error and the value is inverted from the value of the error source node (or the first reachable boolean node to the error source node according to the above P_a explanation). For example, if the error source node has value $true$, $P_{\bar{a}}(n)$ is the probability that the value of n is inverted to $false$.

3. $P_1(n)$ is defined as the probability that the value of node n is $true$, and the value is the same as the value of the corresponding node in the correct timed flow graph.

4. $P_0(n)$ is defined as the probability that the value of node n is $false$, and the value is the same as the value of the corresponding node in the correct timed flow graph.

- Non-Boolean Domain Attributes

  1. $P_p(n)$ is a propagation probability defined as the probability that node n has an error

  2. $P_m(n)$ is a masking probability defined as the probability that node n has no error, i.e., $P_m(n) = 1 - P_p(n)$

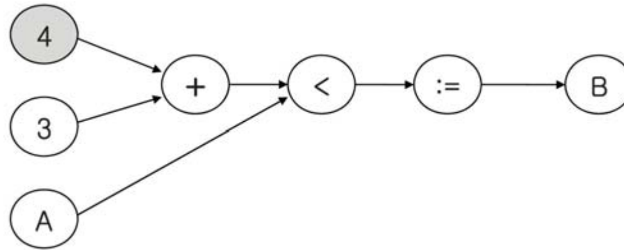


Figure 3.6: Timed data flow graph of expression: $B := A < (4+3)$

A Case Example

Here, we describe our basic idea of computing the error propagation probability using the EPAs. We illustrate the computation process with a simple program as shown in Table 3.3. Suppose this program has a fault on line 1: the correct value is 5 not 4. Given this fault, we try to answer the following question:

Question:

What is the probability that the fault on line 1 can be revealed in the variable C of line 3?

Figure 3.7 represents the timed data flow graph of the program in Table 3.3. We explain the computation of the error propagation probability for each nodes in the timed data flow graph.

1.	$A := (\mathbf{4}+3) \leq X$
2.	$B := A\ \ OR\ \ Y$
3.	$C := true \rightarrow IF\ \ PRE(B)\ \ THEN\ \ true\ \ ELSE\ \ false$

Table 3.3: A simple program

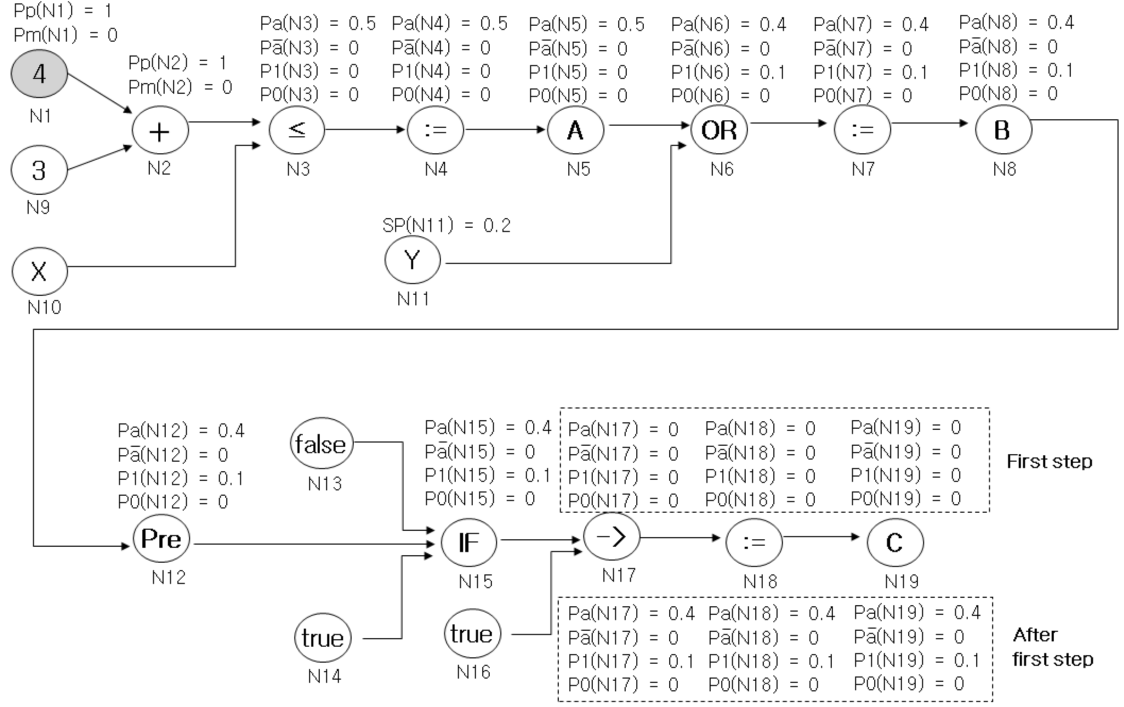


Figure 3.7: Computation of error propagation attributes (EPAs)

1. **N1** : $N1$ is the error source node. The domain of this node is an integer; thus, we use the non-boolean domain attributes for this node. Since we assumed $N1$ has an error, the probability that this node has an error is 1. Thus, $P_p(N1) = 1$ and $P_m(N1) = 1 - P_p(N1) = 0$

2. **N2** : $N2$ is an arithmetic operator node. Arithmetic operators do not have any masking effect. This means that the error in the predecessor nodes will always propagate to the successor nodes of the arithmetic operator node. Thus, $P_p(N2) = 1$ and $P_m(N2) = 0$, as in $N1$.

3. **N3** : $N3$ is a relational operator node. The domain of the relational operator node is boolean. Thus, this node has the boolean domain attributes (i.e.,

$P_a, P_{\bar{a}}, P_1, P_0$). Note that this node is the first reachable boolean domain node from the error source node. Thus, $P_a(n)$ will refer to the value of this node and $P_{\bar{a}}(n)$ will refer to the inverse value of this node. Suppose that the value of $N3$ is $false$. Now, $P_a(N3)$ is bounds to be $false$ and $P_{\bar{a}}(N3)$ is bound to be $true$.

In node $N3$, $P_a(N3)$ is 0.5. That is, there is a 50the error source node will propagate to $N3$. With a 50% chance, conversely, it will "vanish" in $N3$. This probability can be computed as follows. In the expression '$(4+3) \leq X$', we have no knowledge at the value of X. If the value of X was 7, then the evaluation result should be $true$. On the other hand, if there was no error in this expression, the evaluation turns out to be $false$. Thus, $N3$ will have an error if the value of X is 7. However, if the value of X is any greater value, say 100, results of both the error and correct cases are $true$. Thus, the error is masked out. In general, for any expression $X \leq Y$, if we have no information about the values of X and Y, the probability that an error in the expression differentiates the output of the expression from the output of the correct expression is 0.5. Thus, $P_a(N3) = 0.5$ in this example.

Since $N3$ is the first reachable boolean node from the error source node, the other EPAs (i.e. $P_{\bar{a}}, P_1$, and P_0) are set to zeros by this rule.

4. **N4,N7,N18**: $N4, N7$ and $N17$ are assignment operator nodes. Since the assignment operator has no masking effect, the error in the predecessor node will always propagate to the successor node of the assignment operator node. For example, suppose variable B has an error in expression '$A := B$'. Then, A will also have the error. Thus, these assignment operators do not make any changes to the EPAs from the ones coming from the predecessors. We discuss the propagation characteristics of the unary operator including the assignment operator in Section 3.4.2.

5. **N5,N8,N19**: $N5, N13$ and $N18$ are variable nodes. Since only operators can have masking effects, a variable itself does not have any masking effect. Thus, the EPAs of a *variable* node are the same as the ones of its predecessor.

6. **N6**: $N6$ is a logical operator node. The masking effect of a logical operator node depends on the signal probability of the predecessor nodes. For example, suppose variable node A has an error in expression '$A \quad OR \quad B$'. If the value of node B is $true$, then the value of node OR becomes $true$, regardless of the value of node A. The error in node A is masked out in this case. If the value of node B is $false$, then the value of node OR is determined by the value of node A. Thus, the probability of the error in node A to propagate to a successor node depends on the probability of the value of node B being true. In other words, the propagation probability of the error in node A depends on the signal probability of node B. If the signal probability of node B is high, then the propagation probability of the error in node A will be low. In our case example, the signal probability of $N11$ is assumed to be 0.2. Then, $P_a(N6)$ is computed by $P_a(N6) = P_a(N5) \times (1 - SP(N11))$. This means that $N6$ has an error only when $N5$ has an error and the value of $N11$ is $false$.

   $P_1(N6)$ can be computed as follows. Recall that $P_1(N6)$ is a probability that the error is masked out and the value of $N6$ is $true$. There are two cases where the error is masked out and the value of $N6$ is $true$. First, if the value of $N11$ is $true$, then the error in $N5$ is masked out and the value of $N6$ becomes $true$. Second, if (1) the error has been already masked out in $N5$, (2) the value of $N5$ is $true$ and (3) the value of $N11$ is $false$, then the error is masked out in $N6$ and the value of $N6$ becomes $true$. So, $P_1(N6) = (P_a(N5) + P_{\bar{a}}(N5) + P_1(N5) + P_0(N5)) \times SP(N11) + P_1(N5) \times (1 - SP(N11)) = 0.1$. We will discuss the error propagation for logical operators in detail in section 3.4.6.

Since we found no negation operator along the path, $P_{\bar{a}}(N6)$ remains zero. P_0 does not have any change with an OR operator, thus $P_0(N6)$ retains zero as well. We will discuss the error propagation for logical operators in detail in Section 3.4.6.

7. **N12**: $N12$ is the time delay operator PRE. As mentioned in section 3.3.4, the PRE operator is used to refer to the one-step previous value of a node. Suppose the current time step is 10. Then, $PRE(B)$ refers to the value of B in time step 9. If variable B has an error in time step 9, $PRE(B)$ has the same error. As a result, the error propagation attributes of $N12$ is the same as the EPAs of the predecessor node in the previous time step.

8. **N15**: $N15$ is the If_then_else operator node. The masking effect of this operator depends on the location of the error. This operator node has three predecessor nodes (i.e., $C_{PRE}(N15), E_{1PRE}(15)$ and $E_{2PRE}(N15)$). The error propagation probability of this operator node varies by the error location among three predecessor nodes. We discuss this more in detail in section 3.4.5.

   In this case example, $N15$ has an error in the $C_{PRE}(N15)$ node. If the $C_{PRE}(N15)$ node has an error, then $N(15)$ will have a value of a different predecessor node. If $N15$ was supposed to have a value of node $E_{1PRE}(15)$ without the error in the $C_{PRE}(N15)$, $N15$ will instead have a value of $E_{2PRE}(N15)$ with the error. Unless $E_{1PRE}(N15)$ and $E_{2PRE}(N15)$ evaluate to the same result, this error will always propagate to the $N15$. That is, $N15$ has no masking probability in this case. Thus, $N15$ has the same EPAs as $C_{PRE}(N15)$.

9. **N17**: $N17$ is the variable initialization operator node. This operator is used to initialize a variable in time step 1 and is always used with the PRE operator, for example, $C := true \rightarrow PRE(B)$. In this expression, variable C will be $true$

as the initialization value at time step 1, and in time step 2, it will have the value of B in time step 1. The delay effect of this operator affects the error propagation probability of the operator node. For example, suppose variable node B has an error in time step 1. At time step 1, variable node C has the initialization value; thus, the error of node B will not propagate to node C. The error in node B will propagate to node C at time step 2 instead. Thus, $N17$ has two types of EPAs. One is the EPAs at time step 1; there is no error propagated to $N17$ at this time. The other is the EPAs after time step 1; the error in the predecessor nodes will be fully propagated to this node. We will discuss this in section 3.4.2.

We illustrated the basic idea of the error propagation analysis with EPAs. First, we assume there is a fault in the timed data flow graph. From the fault node, we traverse all reachable ancestor nodes and compute the EPAs for each node along the error path. P_p in the non-boolean domain node and $P_a + P_{\bar{a}}$ in the boolean domain node represents the error propagation probability from the error source node to that node. In the beginning of this section, we raised a question about the error propagation probability in the case example. Now, we can obtain the answer based on our error propagation analysis technique as follows:

Question:

What is the probability that the fault on line 1 *can be revealed in the variable* C *of line* 3*?*

Answer:

The probability that the fault of line 1 *can be revealed in variable* C *of line* 3 *is* 0 *at time step* 1 *and* 0.4 *from time step* 2.

The remainder of this section is devoted to a detailed explanation of how we can compute the error propagation probability for each operator in our system model.

3.4.2 Unary Operator Nodes

Error propagation probabilities for unary operator nodes can be computed as follows.

- *Unary_minus*: This operator has no masking effect. For example, if the variable A has an error, $-A$ will also have an error. Thus, the errors in a predecessor node will transparently propagate to a successor node.

- *Assignment*: This operator has no masking effect. That is, the operator just transfers the values of EPAs in a predecessor node to its successor node.

- *NOT*: This operator transfers EPAs in a predecessor to its successor in an *inverse* way. Specifically, the P_a value of the predecessor node will be transferred to the $P_{\bar{a}}$ value of the successor node and vice versa. Likewise, the P_1 value of the predecessor node will be set to the P_0 value for the successor node and vice versa.

- *PRE*: A *PRE* operator node does not have any masking effect by itself. However, when this operator node is used with a variable initialization node, the variable initialization node will mask out the error in the *PRE* operator node for only one time step. Unlike other operators whose masking effect mainly depends on operator types, the masking effect of a *PRE* operator node depends on *time*. We call this kind of masking effect a **temporal masking effect**.

  As mentioned, with an initialization operator node, a *PRE* operator can delay the effect of a computation for one time step. With multiple *PRE* operators, it is possible to delay the effect of a computation for more than a single time

step. Table 3.4 shows an example of how to delay multiple time steps. Suppose the value '4' in line 4 is a fault. Table 3.5 shows the relevant values of the variables at each time step. It takes 4 time steps for the error in variable D to propagate to variable A. If the number of evaluations in this example is shorter than 4, the evaluations would terminate before the error propagates to the longest delayed variable node. If this occurs, the masking probability of the variable node which is delayed less than the evaluation number will be 0, although the masking probability of the variable node which is delayed more than the evaluation number is 1.

1. $A := 1 \rightarrow PRE(B)$
2. $B := 1 \rightarrow PRE(C)$
3. $C := 1 \rightarrow PRE(D)$
4. $D := \mathbf{4}$

Table 3.4: Delayed error propagation

Variable	**Step1**	**Step2**	**Step3**	**Step4**
A	1	1	1	4
B	1	1	4	4
C	1	4	4	4
D	4	4	4	4

Table 3.5: Error propagation time of the example in Table 3.4

Table 3.6 summarizes formulas to compute the error propagation probability for unary operator nodes. Let x be a unary operator node and y be a predecessor node reachable from the error source node.

Operator	Error Propagation Attributes	
	Non-boolean domain	Boolean domain
$:=, PRE$	$P_p(x) = P_p(y), P_m(x) = P_m(y)$	$P_a(x) = P_a(y), P_{\bar{a}}(x) = P_{\bar{a}}(y)$ $P_1(x) = P_1(y), P_0(x) = P_0(y)$
Not	Not applicable	$P_a(x) = P_{\bar{a}}(y), P_{\bar{a}}(x) = P_a(y)$ $P_1(x) = P_0(y), P_0(x) = P_1(y)$
$-$	$P_p(x) = P_p(y), P_m(x) = P_m(y)$	Not applicable

Table 3.6: Formula to compute error propagation attributes for a unary node

3.4.3 Arithmetic Operator Nodes

The arithmetic operator node has a low masking probability, which means that an error in the predecessor node is quite likely to pass to this operator node. Yet, sometimes, arithmetic operator nodes mask out some errors in the predecessor nodes. For example, a multiplication operator node can mask the error of a predecessor node if the value of the other predecessor node is 0 since the value of the multiplication node is zero regardless of the value of the predecessor node with an error. Similarly, a division operator node masks an error of a predecessor node(divisor) if the value of the other predecessor node (dividend) is 0. Nonetheless, these exceptions are fairly limited; thus, the masking probability can be regarded as negligible. Additionally, addition and subtraction operator nodes do not have the masking probability. The only case when they can have the masking probability is that both operands have errors and they cancel the errors of each other out — but we assumed there is only one fault in the system. We hypothesize that the probability of masking with arithmetic operator nodes is thus negligible for the purposes of this work.

Error Independent Nodes

Computing EPAs for error independent nodes is straightforward. Values of the EPAs of the predecessor node will transparently transfer to the successor node. Table 3.7 shows formulas for computation of the EPAs for an error independent node. Let x be an error independent node and y be a predecessor node reachable from the error source node.

Operators	Error Propagation Attributes
$+, -, *, /$	$P_p(x) = P_p(y), P_m(x) = P_m(y)$

Table 3.7: Computations of EPAs for error independent arithmetic node

Error Dependent Nodes

In an error dependent node, both predecessor nodes must be reachable from the error source node. The error dependent node has an error if one of its predecessors propagates the error to the node. Table 3.8 shows formulas to compute the EPAs for the error dependent node. Let x be an error dependent node, and y and z be predecessor nodes reachable from the error source node.

Operators	Error Propagation Attributes
$+, -, *, /$	$P_p(x) = P_p(y) + P_p(z) - P_p(y)P_p(z)$ $P_m(x) = 1 - P_p(x)$

Table 3.8: Computations of EPAs for an error dependent arithmetic node

3.4.4 Relational Operator Nodes

Relational operator nodes, such as $>, <, \geq, \leq, =$, and $\neq$, have a relatively high masking probability. The masking probability of relational operator nodes depends on the domains of the predecessor nodes.

- Non-boolean Domain

  Equality and inequality operators, such as $==$ and $\neq$, mask out almost all errors in the predecessor nodes. We assume that the masking probabilities of these operator nodes are 1 because any error is always masked out with one of these operator nodes. For example, suppose an expression '$i = 1$' where the domain of variable i is an integer. Any error in i which is not 1 should be masked out in this expression.

  Greater- or less-than operators, such as $>, <, \geq$, and $\leq$, have 0.5 for the masking probability for errors in the predecessor nodes. The underlying rationale of this probability is as follows. Suppose an expression '$i > j$' has an error in i. Without any knowledge of the value of i and j, the probability of errors in the i to propagate to the relational operator node is 0.5. This problem can be reduced to the following problem: if we randomly choose two values for i and j, the probability that the expression becomes *true* is 0.5.

- Boolean Domain

  In the boolean domain, any relational operator nodes have *zero* masking probability. An error in a boolean domain node represents that the node has the inverse value of the correct one. This inversion always affects the value of the relational operator node, causing the error to be caught.

Error Independent Node

The domain of the output of any relational operator node is boolean regardless of the domain of predecessor nodes. For example, the output of the expression '$3 > A$' is boolean while the domain of the operands is an integer. Thus, the EPAs for a non-boolean domain (i.e., P_p and P_m) will change into the boolean domain EPAs (i.e., $P_a, P_{\bar{a}}, P_1$, and P_0) in this expression.

Table 3.9 shows formulas to compute the EPAs for the error independent relational operator nodes. Node x is an error independent node and y is a predecessor node reachable from the error source node. As mentioned above, the masking probability for $<, \leq, >, \geq$ is 0.5.

The values of the EPAs for $=$ in the non-boolean domains are all 0s except $P_0(x)$. This is because every error in the predecessor nodes of $=$ operator node is masked out and the value of the $=$ operator node always becomes $false$. By similar reasoning, the values of the EPAs of $\neq$ in the non-boolean domains are all 0s except $P_1(x)$.

For the $=$ and $\neq$ in the boolean domains, the error in the predecessor node is always propagated to the relational operator nodes. However, the information about error propagation attributes becomes meaningless in the relational operator node since the error propagation attributes of the relational operator node is not determined by those of the predecessor node. For example, in expression $'y = true'$, if y has an error, then the error will be propagated to the relational operator node $=$. However, we do not know the error propagation attributes in the relational operator node. Hence, we use only $P_a(x)$ for the relational operator nodes, $=$ and $\neq$. Other EPAs are all 0s.

<table>
<tr><th rowspan="2">Operator</th><th colspan="2">Error Propagation Attributes</th></tr>
<tr><th>Non-boolean domain</th><th>Boolean domain</th></tr>
<tr><td>$<, \leq, >, \geq$</td><td>$P_a(x) = 0.5P_p(y), P_{\bar{a}}(x) = 0$
$P_1(x) = 0, P_0(x) = 0$</td><td>Not applicable</td></tr>
<tr><td>$=$</td><td>$P_a(x) = 0, P_{\bar{a}}(x) = 0$
$P_1(x) = 0, P_0(x) = 1$</td><td rowspan="2">$P_a(x) = P_a(y) + P_{\bar{a}}(y), P_{\bar{a}}(x) = 0$
$P_1(x) = 0, P_0(x) = 0$</td></tr>
<tr><td>$\neq$</td><td>$P_a(x) = 0, P_{\bar{a}}(x) = 0$
$P_1(x) = 1, P_0(x) = 0$</td></tr>
</table>

Table 3.9: Computations of EPAs for error independent relational node

Error Dependent Node

Let y and z be non-boolean domain predecessors of error dependent node x. Errors in y and z will propagate to x at a half rate as in the case of the error independent node. However, one more consideration is to eliminate any duplicate reception of the same error through y and z in the dependent case. We subtract the probability, $P_p(y) \cdot P_p(z)$ from the computation formula, as shown in Table 3.10. If the domain of both y and z is boolean, x will have an error when exactly one predecessor propagates the error to x. If both predecessors propagate the same error, then the error will be canceled out by the other. The computation formulas are summarized in Table 3.10.

3.4.5 *If_then_else* Operator Nodes

An *if_then_else* operator node n has three predecessor nodes such as $C_{PRE}(n)$, $E_{1PRE}(n)$ and $E_{2PRE}(n)$. The masking effect of n varies depending on which predecessor node has an error and which domain it is. If $C_{PRE}(n)$ has an error, the error will always propagate to n since the error will make the node n to have the value of different pre-

<table>
<tr><th rowspan="2">Operator</th><th colspan="2">Error Propagation Attributes</th></tr>
<tr><th>Non-boolean domain</th><th>Boolean domain</th></tr>
<tr><td>$<, \leq, >, \geq$</td><td>$P_a(x) = 0.5(P_p(y) + P_p(z) - P_p(y)P_p(z))$
$P_{\bar{a}}(x) = 0, P_1(x) = 0, P_0(x) = 0$</td><td>Not applicable</td></tr>
<tr><td>$=$</td><td>$P_a(x) = 0, P_{\bar{a}}(x) = 0$
$P_1(x) = 0, P_0(x) = 1$</td><td rowspan="2">$P_a(x) = P_a(y)(1 - P_a(z))(1 - P_{\bar{a}}(z)) + P_a(z)(1 - P_a(y))(1 - P_{\bar{a}}(y)) + P_{\bar{a}}(y)(1 - P_a(z))(1 - P_{\bar{a}}(z)) + P_{\bar{a}}(z)(1 - P_a(y))(1 - P_{\bar{a}}(y))$
$P_{\bar{a}}(x) = 0, P_1(x) = 0, P_0(x) = 0$</td></tr>
<tr><td>$\neq$</td><td>$P_a(x) = 0, P_{\bar{a}}(x) = 0$
$P_1(x) = 1, P_0(x) = 0$</td></tr>
</table>

Table 3.10: Computations of EPAs for error dependent relational node

decessor nodes. If E_{1PRE} or E_{2PRE} has an error, on the other hand, the propagation probability of the error depends on two conditions: (1) the error in the error source node should propagate to one of the predecessor nodes of n. (2) the predecessor node with an error should evaluate. Node $E_{1PRE}(n)$ evaluates when $C_{PRE}(n)$ evaluates $true$ while $E_{2PRE}(n)$ evaluates when $C_{PRE}(n)$ evaluates $false$. Therefore, the probability of $E_{1PRE}(n)$ to evaluate is $SP(C_{PRE}(n))$ and the probability of $E_{2PRE}(n)$ to evaluate is $1 - SP(C_{PRE}(n))$.

Error Independent Node

As mentioned above, the error propagation probability from the predecessor node to the successor node of if_then_else depends on the error location and the signal probability of the $C_{PRE}(n)$. Since only one predecessor is reachable from the error

source node, the computation process is straightforward, as shown in Table 3.11.

Error Node	Error Propagation Attributes(n : If_then_else node)	
	Non-boolean domain	Boolean domain
x ($C_{PRE}(n)$)	$P_p(n) = P_a(x) + P_{\bar{a}}(x)$ $P_m(n) = 1 - P_p(n)$	$P_a(w) = P_a(x) + P_{\bar{a}}(x)$ $P_{\bar{a}}(n) = 0, P_1(n) = 0, P_0(n) = 0$
y ($E_{1PRE}(n)$)	$P_p(n) = SP(x)P_p(y)$ $P_m(n) = 1 - P_p(n)$	$P_a(n) = SP(x)P_a(y)$ $P_{\bar{a}}(n) = SP(x)P_{\bar{a}}(y)$ $P_1(n) = 0, P_0(n) = 0$
z ($E_{2PRE}(n)$)	$P_p(n) = (1 - SP(x))P_p(z)$ $P_m(n) = 1 - P_p(n)$	$P_a(n) = (1 - SP(x))P_a(z)$ $P_{\bar{a}}(n) = (1 - SP(x))P_{\bar{a}}(z)$ $P_1(n) = 0, P_0(n) = 0$

Table 3.11: Computing EPAs for the error independent *if-then-else* node

Error Dependent Node

The basic idea to compute the EPAs for the error dependent node is not different from the case of the error independent node. If $C_{PRE}(n)$ has an error, the error will always propagate to the If_then_else operator node n regardless of whether $E_{1PRE}(n)$ or $E_{2PRE}(n)$ has an error or not. If $E_{1PRE}(n)$ or $E_{2PRE}(n)$ has an error, the error may propagate to the n depending on the probability that the predecessor node with an error will be evaluated. The detailed formulas for non-boolean nodes and boolean nodes appear in Table 3.12 and 3.13, respectively.

Error Propagation Attributes (n : If_then_else node, x : $C_{PRE}(n)$, y : $E_{1PRE}(n)$, z : $E_{2PRE}(n)$)	
Error Node(s)	Non-boolean domain
x *and* y	$P_p(n) = P_p(x) + SP(x)P_m(x)P_p(y)$ $P_m(n) = 1 - P_p(n)$
x *and* z	$P_p(n) = P_p(x) + (1 - SP(x))P_m(x)P_p(z)$ $P_m(n) = 1 - P_p(n)$
y *and* z	$P_p(n) = P_p(y)P_p(z) + SP(x)P_p(y)P_m(z)$ $+ (1 - SP(x))P_m(y)P_p(z)$ $P_m(n) = 1 - P_p(n)$
x *and* y *and* z	$P_p(n) = P_p(x)P_p(y)P_p(z) + P_m(x)P_p(y)P_p(z)$ $+ (1 - SP(x))P_m(x)P_m(y)P_p(z)$ $+ SP(x)P_m(x)P_p(y)P_m(z)$ $P_m(n) = 1 - P_p(n)$

Table 3.12: Computing EPAs for the error dependent *if-then-else* node (non-boolean node)

3.4.6 Logical Operator Nodes

The masking probability for logical operator node n depends on the signal probability of the predecessor nodes of n as well as the operator type. For example, the masking probability of the OR operator node is proportional to the signal probability of the predecessor nodes, while the masking probability of the AND operator is inversely proportional to the signal probability of the predecessor nodes. On the other hand, the XOR operator has no masking probability. Since we have the masking probability for each logical operator node, we can compute the propagation probability for each

Error Propagation Attributes (n : *If_then_else* node, x : $C_{PRE}(n)$, y : $E_{1PRE}(n)$, z : $E_{2PRE}(n)$)	
Error Node(s)	Boolean domain
x *and* y	$P_a(n) = (P_a(x) + P_{\bar{a}}(x)) + SP(x)(1 - (P_a(x) + P_{\bar{a}}(x)))$ $\quad + (P_a(y) + P_{\bar{a}}(y))$ $P_{\bar{a}}(n) = 0, P_1(n) = 0, P_0(n) = 0$
x *and* z	$P_a(n) = (P_a(x) + P_{\bar{a}}(x)) + SP(x)(1 - (P_a(x) + P_{\bar{a}}(x)))$ $\quad + (P_a(z) + P_{\bar{a}}(z))$ $P_{\bar{a}}(n) = 0, P_1(n) = 0, P_0(n) = 0$
y *and* z	$P_a(n) = (P_a(y) + P_{\bar{a}}(y))(P_a(z) + P_{\bar{a}}(z))$ $\quad + SP(x)(P_a(y) + P_{\bar{a}}(y))(1 - (P_a(z) + P_{\bar{a}}(z)))$ $\quad + (1 - SP(x))(1 - (P_a(y) + P_{\bar{a}}(y)))(P_a(z) + P_{\bar{a}}(z))$ $P_{\bar{a}}(n) = 0, P_1(n) = 0, P_0(n) = 0$
x *and* y *and* z	$P_a(n) = (P_a(x) + P_{\bar{a}}(x))(P_a(y) + P_{\bar{a}}(y))(P_a(z) + P_{\bar{a}}(z))$ $\quad + (1 - (P_a(x) + P_{\bar{a}}(x)))(P_a(y) + P_{\bar{a}}(y))(P_a(z) + P_{\bar{a}}(z))$ $\quad + (1 - SP(x))(1 - (P_a(y) + P_{\bar{a}}(y)))$ $\qquad (1 - (P_a(y) + P_{\bar{a}}(y)))(P_a(z) + P_{\bar{a}}(z))$ $\quad + SP(x)(1 - (P_a(x) + P_{\bar{a}}(x)))$ $\qquad (1 - (P_a(y) + P_{\bar{a}}(y)))(1 - (P_a(z) + P_{\bar{a}}(z)))$ $P_{\bar{a}}(n) = 0, P_1(n) = 0, P_0(n) = 0$

Table 3.13: Computing EPAs for the error dependent *if-then-else* node(boolean node)

operator node as well. We next discuss the error propagation probability for error independent and dependent nodes in the following two sections.

Error Independent Node

The computation of EPAs for the error independent node is primarily based on the signal probability of a predecessor node. As discussed in the beginning part of this chapter, any node reachable from the error source node will have EPAs computed, while the boolean nodes that are unreachable from the error source node have the signal probability computed through Algorithm 1.

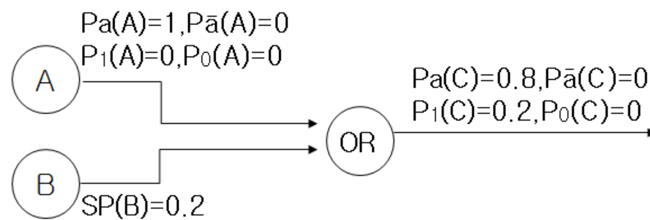


Figure 3.8: An error independent logical node

Figure 3.8 shows an example of an error independent node. In this example, $P_a(OR)$ is computed by a product of the probability of node A to have an error ($P_a(A)$) and the probability of node B to be $false$ $(1 - SP(B))$. Thus, $P_a(OR) = P_a(A)(1-SP(B)) = 0.8$, in the above example. $P_{\bar{a}}(OR)$ is a product of the probability of node A to have an inverse error and the probability of node B to be $false$. Thus, $P_{\bar{a}}(OR) = P_{\bar{a}}(OR)(1 - SP(B)) = 0$.

$P_1(OR)$ is a probability that the error of the error source node is masked out and the value of the OR node is $true$. If the value of node B is $true$, then the value of the OR node becomes $true$, regardless of the value of node A. Thus, the error in node A is masked out and the value of the OR node becomes $true$. The other case, where the error is masked out and the value of the OR node becomes $true$, takes place when the error has already been masked out in node A, and the values of A and B are $true$ and $false$, respectively. Based on these properties, we compute

$P_1(OR) = SP(B) + P_1(A)(1 - SP(B)) = 0.2$. Finally, the only case where the error is masked out and the value of the OR node is $false$ occurs when the value of both predecessor nodes of the OR node are $false$, meaning that the error has been masked out in node A and the value of node A is $false$, and the value of node B is also $false$ (i.e., $(1 - SP(B))$). Thus, $P_0(OR) = P_0(A)(1 - SP(B)) = 0$ in this case.

Similarly, EPAs for other logical operator nodes can be computed as shown in Table 3.14. In the table, r represents a fault probability of the error source node in the boolean domain, or $P_a(x)$,where x is the first boolean node reachable from the error source node if the error source node is in the non-boolean domain. In Table 3.14, x is an error independent node, y is a predecessor node of x unreachable from the error source node, and z is another predecessor node of x reachable from the error source node.

Error Dependent Node

An error dependent node has some predecessor nodes that are all reachable from the error source node. Since any node that is reachable from the error source node has its EPAs, the error dependent node has more than one predecessor node with their own EPAs. In Figure 3.9, nodes D and E are error independent nodes since they have only one predecessor node reachable from the error source node. On the other hand, node F is an error dependent node because both predecessor nodes are reachable from the error source node, thus both predecessors have their EPAs. Combining the EPAs of the two nodes is different depending on the type of the error dependent node.

In Figure 3.9, $P_a(F)$ needs to be computed when both predecessors have the same type of the error as the error source node $(i.e., the P_a$ attribute) or one predecessor has the same type of error as the error source node and the other is $true$; $P_a(F) = 0$ for the other case. Suppose both predecessors have errors and the values are $true$. Then,

Operators	Error Propagation Attributes(r: fault probability)
AND	$P_1(x) = P_1(z)SP(y)$ $P_a(x) = P_a(z)SP(y)$ $P_{\bar{a}}(x) = P_{\bar{a}}(z)SP(y)$ $P_0(x) = r - [P_1(x) + P_a(x) + P_{\bar{a}}(x)]$
OR	$P_1(x) = r - [P_0(x) + P_a(x) + P_{\bar{a}}(x)]$ $P_a(x) = P_a(z)(1 - SP(y))$ $P_{\bar{a}}(x) = P_{\bar{a}}(z)(1 - SP(y))$ $P_0(x) = P_0(z)(1 - SP(y))$
XOR	$P_1(x) = r - [P_0(x) + P_a(x) + P_{\bar{a}}(x)]$ $P_a(x) = P_{\bar{a}}(z)SP(y) + P_{\bar{a}}(z)(1 - SP(y))$ $P_{\bar{a}}(x) = P_a(z)SP(y)) + P_a(z)(1 - SP(y))$ $P_0(x) = P_1(z)SP(y) + P_0(z)(1 - SP(y))$

Table 3.14: Computation of EPAs for an error independent node

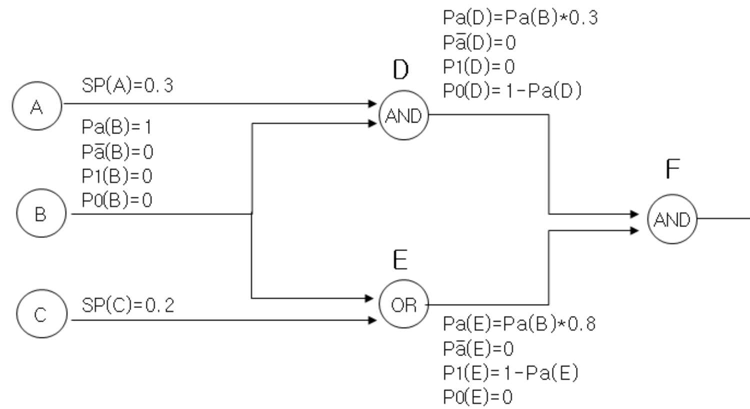


Figure 3.9: An error dependent node

the output of the AND operator should be $true$. This means that the correct values for both predecessors are $false$ and the output of the AND node should be $false$. Since the actual output is different from the correct one, the error has propagated to the output of the AND operator. Thus, we compute $P_a(F) = P_a(D)P_a(E) + P_a(D)P_1(E) + P_1(D)P_a(E)$. Computation of $P_{\bar{a}}(F)$ is the same as $P_a(F)$ except using $P_{\bar{a}}$ instead of P_a. Thus, $P_{\bar{a}}(F) = P_{\bar{a}}(D)P_{\bar{a}}(E) + P_{\bar{a}}(D)P_1(E) + P_1(D)P_{\bar{a}}(E)$.

$P_1(F)$ is computed in the AND node only when both predecessor nodes are $true$ and the error from the error source node has been masked out in both predecessors. Thus, $P_1(F) = P_1(D)P_1(E)$. Finally, $P_0(F)$ is obtained when one of the predecessors is $false$, or one predecessor has a P_a attribute and the other has a $P_{\bar{a}}$ attribute. For example, suppose node D has a $P_a(D)$ and node E has its inverse value from node D (i.e., $P_{\bar{a}}(E)$). The output will be $false$ and the error is masked out since the output of the AND node will also be $false$ with the correct inputs. Thus, $P_0(F) = P_0(D)(P_a(E) + P_{\bar{a}}(E) + P_1(E) + P_0(E)) + P_0(E)(P_a(D) + P_{\bar{a}}(D) + P_1(D)) + P_a(D)P_{\bar{a}}(E) + P_{\bar{a}}(D)P_a(E)$. Table 3.15 summarizes the computation of EPAs for error dependent nodes.

In Figure 3.10, there can be 16 combinations in computing EPAs for an AND operator with the EPAs of two predecessor nodes. However, in reality, only 4 combinations are allowed in computation, including $P_a(A)P_a(A), P_{\bar{a}}(A)P_{\bar{a}}(A), P_1(A)P_1(A)$, and $P_0(A)P_0(A)$ because the EPAs of one predecessor determines the EPAs of the other predecessor. Other combinations, such as $P_a(A)P_{\bar{a}}(A), P_aP_1(A)$, etc, are not possible. For example, $P_a(A)P_{\bar{a}}(A)$ is impossible since node A cannot have an error value and its inverse value at the same time. To identify such impossible (or impractical) combinations of EPAs, we employ symbol representations instead of numbers for EPA values, as in the computation of signal probability discussed in 3.3.

We take the same strategy to compute the EPAs for error dependent nodes. The

Operators	Error Propagation Attributes(r: fault probability)
AND	$P_0(x) = r - [P_1(x) + P_a(x) + P_{\bar{a}}(x)]$ $P_a(x) = P_1(y)P_a(z) + P_a(y)P_1(z) + P_a(y)P_a(z)$ $P_{\bar{a}}(x) = P_1(y)P_{\bar{a}}(z) + P_{\bar{a}}(y)P_1(z) + P_{\bar{a}}(y)P_{\bar{a}}(z)$ $P_1(x) = P_1(y)p_1(z)$
OR	$P_0(x) = P_0(y)P_0(z)$ $P_a(x) = P_0(y)P_a(z) + P_a(y)P_0(z) + P_a(y)P_a(z)$ $P_{\bar{a}}(x) = P_0(y)P_{\bar{a}}(z) + P_{\bar{a}}(y)P_0(z) + P_{\bar{a}}(y)P_{\bar{a}}(z)$ $P_1(x) = r - [P_0(x) + P_a(x) + P_{\bar{a}}(x)]$
XOR	$P_0(x) = P_a(y)P_a(z) + P_{\bar{a}}(y)P_{\bar{a}}(z) + P_1(y)P_1(z) + P_0(y)P_0(z)$ $P_a(x) = P_{\bar{a}}(y)P_1(z) + P_1(y)P_{\bar{a}}(z) + P_a(y)P_0(z) + P_0(y)P_a(z)$ $P_{\bar{a}}(x) = P_a(y)P_1(z) + P_1(y)P_a(z) + P_{\bar{a}}(y)P_0(z) + P_0(y)P_{\bar{a}}(z)$ $P_1(x) = r - [P_0(x) + P_a(x) + P_{\bar{a}}(x)]$

Table 3.15: Computation of EPAs for error dependent node (r: fault probability)

EPAs of the error source node will be maintained during the computation of EPAs for each node. In other words, we assign a character for each error propagation attribute of the error source node and keep this character during the computation of the EPAs for all the nodes reachable from the error source node. After the EPAs of every node are obtained, we remove impossible combinations and suppress the exponents by Theorem 2 and Corollary 1. Finally, the character is replaced by the value of the error propagation attribute of the error source node. We omit the proofs of the theorem and corollary, but they can be done similarly with the proof of Theorem 1.

Theorem 2 *A product of the same error propagation attributes of a node is equal to the single error propagation attribute of that node, i.e.,* $P_a(v)P_a(v) = P_a(v)$.

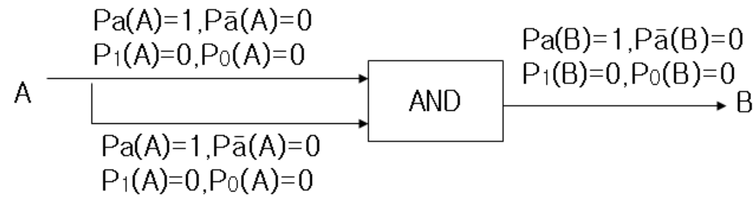


Figure 3.10: A simple error dependent node

Corollary 1 *A product of any different error propagation attributes of a node is* 0*, i.e.,* $P_a(v)P_1(v) = 0$.

3.4.7 Error Propagation Probability of a Variable with a Feedback Input

In section 3.3.4, we showed the signal probability of a variable with a feedback input changes by test case length. In this section, we discuss the error propagation probability of a variable that has a feedback input using an example of Figure 3.11. Since the error propagation probability of an independent logical operator depends on the signal probability, the error propagation probability also changes by the test case length. However, an independent node becomes a dependent node from time step 2 if the node has a feedback input. Figure 3.11 show the case where an independent node change to a dependent node at time step 2. Suppose variable A is reachable to an error source node. Then, variable A will have the error propagation attributes and the signal probability of B will be initially 1 since its initial value is *true*. Hence, the error propagation attributes of B is same with A at first time step. At second time step, variable B will also have error propagation attributes as shown in the figure. Thus, it becomes a dependent node from the second time step and the the error propagation probability of B can be computed by the formula in Table 3.15.

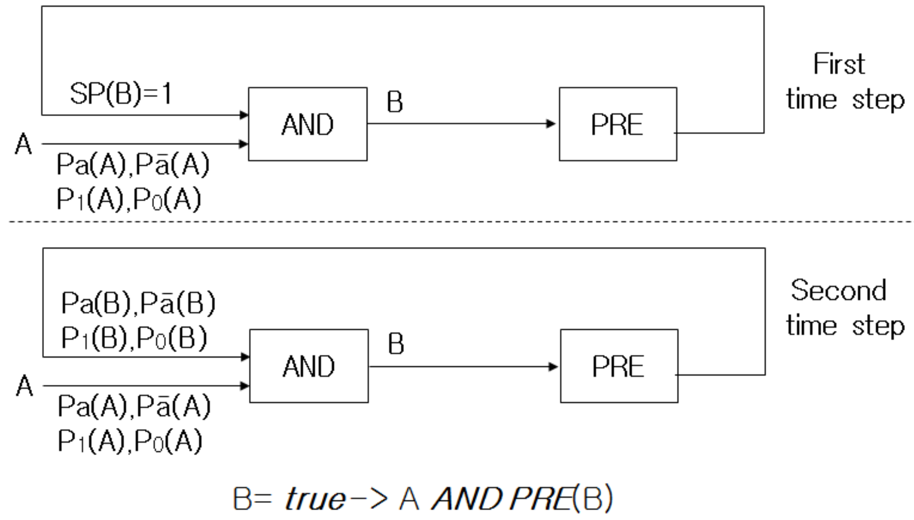


Figure 3.11: A variable of which output is fed back

3.4.8 Algorithm of Computing Error Propagation Probability

We illustrated the basic idea of the error propagation analysis with EPAs, and explained how we can compute error propagation probability for each operator in our system model. Putting it all together, Algorithm 2 describes the overall computation process of the error propagation probability for the timed data flow graph.

Algorithm 2 Computation of Error Propagation Probability

1: Let v be the error source node.
2: Assign a character representing the error propagation attributes for v.
3: **for** the number of evaluation **do**
4: Compute signal probability for all boolean domain node.
5: **for** each node u that is reachable from v **do**
6: Compute the error propagation attributes of u with a symbolic representation.
7: **end for**
8: **for** the symbolic representation for each node u **do**
9: Simplify the symbolic representation for error propagation attributes by Theorem 2 and Corollary 1.
10: Replace characters with numbers and evaluate.
11: **end for**
12: **end for**

Chapter 4

Oracle Data Selection

In this chapter, we introduce our oracle data selection heuristics using the error propagation analysis. The underlying idea of the oracle data selection heuristics is to sort all nodes with respect to their fault finding capabilities and choose oracle data from this order. To compute the fault finding capability of each node, we use the error propagation analysis described in Chapter 3. Before we introduce the oracle data selection heuristics, we first describe the terms and parameters for the oracle data selection heuristics.

4.1 Definition of Terms

We first define **ranking** and **rank**, which is a basic notion for our oracle data selection heuristics.

Let S_n be a set of lists generated by permuting n objects.

Definition 25 ***Ranking:***
*A **ranking** of n objects, $\alpha \in S_n$ is an ordered list consisting of the objects.*

Definition 26 ***Rank:***
*The **rank** of object k in α, $\alpha(k)$ is the distance from the first object of α to k.*

According to the definitions above, a ranking is an ordered list of objects and a rank

is the place of an object in a ranking. We also formally define some well-known terms in the testing community: test input, test case and test case length.

Let $IV = \{I_1, I_2, ..., I_n\}$ be the set of input nodes in a system.

Definition 27 ***Test input:***
*A **test input** is the set of values,* $t_1 = \{v_1, v_2, ..., v_n | v_i \in Domain(I_i)\}$.

Definition 28 ***Test case:***
*A **test case** is a finite sequence of test input.*

Definition 29 ***Test case length:***
*The **test case length** is the length of sequence of a test case.*

Next we will describe some parameters for our oracle data selection heuristics.

4.2 Parameters for the oracle data selection heuristics

As mentioned above, our oracle data selection heuristics are based on the error propagation analysis of a system which needs the values of certain parameters to be determined in advance. When the predetermined values of the parameters are not correct with respect to the values used in real testing, the correctness of our oracle data selection heuristics will be compromised. We will show the experimental results investigating the effect of an incorrect value for these parameters in Section 6.4.

4.2.1 Signal Probability

The signal probability of boolean input nodes have a critical effect on the error propagation probability during the error propagation analysis. The signal probability of input nodes first affects the masking probability of logical operator nodes and – as a result– it can affect the error propagation probability in a system. Thus, to perform

the error propagation analysis, the signal probability for each input node has to be predetermined. To obtain the precise error propagation probability in a system, the predetermined signal probability has to correspond to the signal probability of the test suites that will be used during the real testing.

4.2.2 Fault Distribution

In addition to signal probability, the fault distribution over a system also has an important impacts on our oracle data selection heuristics. If a specific node has a high probability of having a fault, all nodes that are computationally connected to the fault node will have a relatively high failure probability. Similarly, if a node has a low probability to have a fault, every node connected to this node will have a low failure probability. Thus, the probability of each node having a fault is an important factor for our error propagation analysis. The assumed fault distribution over the nodes of a system has to be determined before the error propagation analysis begins. We may obtain the probability through the analysis of a programmer's tendency to make mistakes. If we have no prior information regarding the programmer's fault-making tendency, we can assume that every node in a system can have the same error probability.

4.2.3 Test Case Length

The test case length has two significant effects on the error propagation probability in a system. First, the test case length affects the signal probability of logical operator nodes that have feedback input as described in Section 3.3.4. If an operator node has a feedback input, the signal probability might change depending on the number of evaluations for the node. From a testing point of view, the number of evaluations is determined by the test case (more specifically, the test case length). Since the error

propagation probability of a node is affected by the signal probability, the test case length can also affect the error propagation probability of a node.

Second, the test case length has an effect on the temporal masking effect as described in Section 3.4.2. A Pre operator node delays the error propagation to its guarded variable one time step. If a Pre operator node appears several times in one computational path, they can delay the error propagation as many time steps as the number of Pre operator nodes in the computational path. If the test case length is shorter than the number of Pre operator nodes in one computation path, the errors cannot propagate to the nodes that are located in the latter part of the computation path.

To sum up, the test case length plays a critical role in our error propagation analysis. Thus, it has to be determined before the error propagation analysis is performed. This length can be obtained by investigating the test suites that will be used during the real testing.

4.3 Error Propagation Table

In this section, we explain the construction of the error propagation table, which is foundational work for our oracle data selection heuristics. The error propagation table is a matrix containing the error propagation probability for every node in the timed data flow graph. The formal definition of the graph is as follows.

Definition 30 ***Error Propagation Table:***
Let N be the number of nodes in the timed data flow graph $\mathcal{G}$. The error propagation table M of $\mathcal{G}$ is $N \times N$ matrix, where $M(x, y)$ represents the error propagation probability from x to y, that is, $PP(x, y)$.

To construct the error propagation table, we need the three parameters for the error propagation analysis to be determined as described in the previous section.

After the values of the three parameters are determined, we can perform the error propagation analysis for every node in the timed data flow graph. That is, every node in the graph becomes an error source node once, and the error propagation attributes of each node that is reachable from the error source node will be computed. As an example, Figure 4.1 shows a timed data flow graph with the predetermined signal probability for the input nodes. Suppose every node in the graph has the same fault probability (i.e., r), and the test case length is greater than or equal to 1. Then, the error propagation table for this timed data flow graph is shown in Table 4.1.

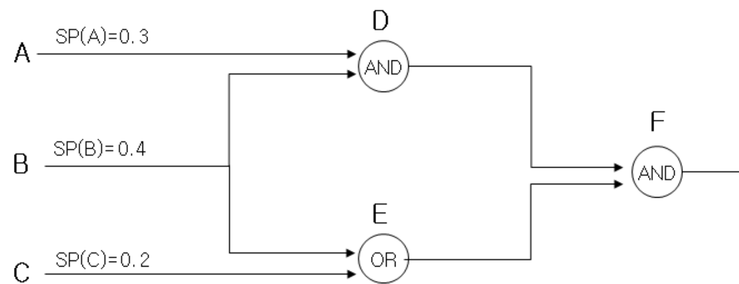


Figure 4.1: A timed data flow graph with the predetermined signal probability for input nodes

The algorithm to construct an error propagation table appears in Algorithm 3. The time complexity to construct the error propagation table is $O(n^2)$ where n is the number of nodes in the timed data flow graph.

	A	B	C	D	E	F
A	r	0	0	$0.4r$	0	$0.208r$
B	0	r	0	$0.3r$	$0.8r$	$0.3r$
C	0	0	r	0	$0.6r$	$0.072r$
D	0	0	0	r	0	$0.52r$
E	0	0	0	0	r	$0.12r$
F	0	0	0	0	0	r

Table 4.1: Error propagation table for the timed data flow graph in Figure 4.1

Algorithm 3 Construction of an Error Propagation Table

1: **for** each node, v in the timed data flow graph G **do**
2:     **for** each node, u in G that is reachable to v **do**
3:         Compute the error propagation probability from v to u, that is, $PP(v, u)$.
4:         Fill $M(v, u)$ with $PP(v, u)$.
5:     **end for**
6: **end for**

4.4 Oracle Data Selection Heuristics: Absolute Ranking vs. Relative Ranking

In this section, we introduce two oracle data selection heuristics: absolute ranking and relative ranking. Both heuristics are computed from the error propagation table, but they have a different view of the repeated detection of an error. The following is the definition of the repeated detection of errors.

Definition 31 ***Repeated detection:***
*A **repeated detection** of an error is that the error is caught by more than one oracle datum*

From a test effectiveness perspective, the repeated detection of an error does not increase the test efficiency. Rather, it may waste test resources. Thus, oracle data have to be chosen in a way to reduce the repeated detection of errors among oracle data. In relative ranking, the oracle data are chosen to minimize the repeated detection of errors, but it takes a longer time to generate the ranking. Meanwhile, in absolute ranking, the oracle data are chosen according to their error catching probability, but the repeated detection of errors is not considered. In the following sections, these two heuristics are explained in detail.

4.4.1 Absolute Ranking

The goal of absolute ranking is to choose oracle data which will show the best performance in catching errors. In this heuristic, the oracle data selection is based on the failure probability of each node. Recall that the failure probability of node n is a conditional probability such that *Pr(n has an error | n is monitored).* The failure probability of nodes is computed from the error propagation table using Algorithm 4. The time complexity of the algorithm to make an absolute ranking from a timed data flow graph is $O(n^2)$, where n is the number of nodes in the timed data flow graph.

Algorithm 4 Compute the failure probability (FP) of each node in a system

1: Let M be an $N \times N$ error propagation table.
2: **for** each column j of error propagation table(M) **do**
3: $FP(j) = 0$.
4: **for** each raw i of error propagation table(M) **do**
5: $FP(j) = FP(j) + M(i, j)$.
6: **end for**
7: **end for**

Intuitively, the failure probability of a node is the sum of all error propagation

probabilities from all reachable nodes to the node. Thus, if a node has a great failure probability, the node has a high probability to catch errors. Table 4.2 shows the computation of the failure probability from the error propagation table of Table 4.1

	A	B	C	D	E	F
A	r	0	0	$0.4r$	0	$0.208r$
B	0	r	0	$0.3r$	$0.8r$	$0.3r$
C	0	0	r	0	$0.6r$	$0.072r$
D	0	0	0	r	0	$0.52r$
E	0	0	0	0	r	$0.12r$
F	0	0	0	0	0	r
FP	**r**	**r**	**r**	**1.7r**	**2.4r**	**2.22r**

Table 4.2: Failure probability of variables in Table 4.1

An absolute ranking is a ranking of nodes that are arranged by the failure probability of the nodes. For example, the absolute ranking of nodes in Table 4.2 is $< E, F, D, C, B, A >$. If we choose oracle data from the absolute ranking in order, the chosen oracle data will show the good performance to catch errors in terms of the absolute number of faults found.

4.4.2 Relative Ranking

As mentioned above, the repeated detection of an error among the oracle data is not considered in oracle data selection by absolute ranking. In the worst case, even though each oracle datum has a high potential to catch errors, the overall test effectiveness does not increase. Therefore, we suggest another oracle data selection heuristic, *relative ranking*, which aims to maintain high effectiveness in fault finding while reducing

repeated detection. In the relative ranking heuristic, if a node has a high potential of catching "new errors" (i.e., they cannot be caught from the already selected oracle data), the node will be chosen as an oracle datum. The computation of the relative ranking also starts with the error propagation table. In the error propagation table, the node with the highest failure probability is chosen as the first member of the relative ranking. Let us call the node, N_1. After choosing N_1, we recompute the failure probability of every other node by subtracting the probability of repeated detection of errors from the failure probability of every other node. For example, suppose the failure probability of node N_2 is σ and the probability of repeated detection of errors between N_1 and N_2 is τ. Then, the new failure probability of N_2 is $\sigma - \tau$.

Once the failure probability for every node (except the nodes which are already chosen in the ranking) is recomputed, the node with the highest failure probability is chosen as the next member of the relative ranking. This greedy process is repeated until every node is selected in the relative ranking. In this way, oracle data selection by relative ranking can reduce the repeated detection of errors, and at the same time, it can increase the fault finding capability of the oracle data. The relative ranking is computed by Algorithm 5. A more detailed algorithm for relative ranking is shown in the Appendix A. The time complexity for the algorithm is $O(n^3)$. Recall that the time complexity of the algorithm to make an absolute ranking is $O(n^2)$. Relative ranking requires a more expensive process than absolute ranking, though it can give a better solution to increase test effectiveness than the absolute ranking.

Algorithm 5 Compute relative ranking from a timed data flow graph

1: **for** the number of nodes of a timed data flow graph **do**
2: Construct an error propagation table.
3: Compute the failure probability of nodes by subtracting the probability of the repeated detection of errors from the failure probability of the nodes.
4: Choose a node with the highest failure probability as a new member of the relative ranking and locate it in the last position of the ranking.
5: **end for**

4.5 Oracle Data Selection Heuristics: Interstate Check vs. Last Step Check

As mentioned in Section 4.2.3, the test case length affects the signal probability of logical operator nodes that have feedback inputs, and the change of signal probability causes the error propagation probability of nodes to change. Consequently, the failure probability of a node can change at each evaluation step. This means that a ranking which is based on the failure probability can change in every evaluation step. Then, a question arises. *Which ranking will we be used for oracle data selection?* Depending on the evaluation time of nodes, our oracle data selection heuristics are distinguished by *the interstate check* and *last step check*.

4.5.1 Interstate Check

For the interstate check, a ranking is generated by the relative ranking heuristic, but the failure probability of nodes (which is a critical factor for the rank of the nodes) is evaluated from all steps. Thus, every node has a failure probability of as many as the number of evaluation steps in the interstate check. Among the failure probabilities of nodes for every evaluation step, the node with the highest failure probability is

chosen for the member of the ranking. Then, the failure probability of other nodes for every evaluation step is recomputed with consideration of the repeated detection of errors. For example, Table 4.3 shows the failure probabilities of four nodes for three evaluation steps. Since the failure probability of node 3 at step 3 is the highest, node 3 is chosen as the first member of the ranking with the interstate check. For the next step, the failure probability of every node (except node 3) for every step is recomputed with consideration of repeated detection of errors with node 3. This process is repeated until all nodes are located in the ranking.

The ranking using the interstate check is computed by Algorithm 6. For a real test environment, if we can monitor the intermediate state of a system during testing, the interstate check will give a better indication to predict the fault finding capability of nodes.

	Node1	**Node2**	**Node3**	**Node4**
Step 1	$11r$	$14r$	$23r$	$16r$
Step 2	$13r$	$22r$	$25r$	$11r$
Step 3	$15r$	$22r$	$27r$	$10r$

Table 4.3: Failure probabilities of nodes for three evaluation steps

4.5.2 Last Step Check

For the last step check heuristic, a ranking is also generated by the relative ranking. However, unlike the interstate check, the failure probability of nodes is evaluated only at the last step. This heuristic is useful for the systems where the intermediate states of the systems can not be evaluated, and only the last state can be monitored. In this heuristic, the errors which will propagate to the last state contribute to the failure probability of nodes. Thus, the oracle data selection by this heuristic can show

Algorithm 6 Compute a ranking for interstate check from a timed data flow graph

```
1: for  the number of nodes of a timed data flow graph do
2:     for the number of evaluation steps(test case length) do
3:         Construct an error propagation table.
4:         Compute the failure probability of nodes by subtracting the probability of the
           repeated detection of errors from the failure probability of the nodes.
5:     end for
6:     Choose a node with the highest failure probability as a new member of the relative
       ranking and locate it at the last position of the ranking.
7: end for
```

the best performance in error catching on the last state of a system. The algorithm to generate the ranking for the last step check is the same as the algorithm for the relative ranking except that the failure probability is evaluated from the last step.

Chapter 5

Implementation

This chapter describes our implementation for the oracle data selection criterion based on the error propagation analysis. The implementation generates a ranking of nodes that can be used for oracle data selection.

5.1 System Overview

In this section, we introduce the main components of our implementation.

5.1.1 System Architecture

Our implementation consists of four major components:

- Translator, which translates a Lustre model into an Error Propagation Tree.
- Signal Probability Analyzer that computes the signal probability of every logical node.
- Error Propagation Analyzer that computes the error propagation probability for every node in the error propagation tree.
- Fault Model Generator that provides different fault models.

Figure 5.1 shows a component diagram of how these components interact with each other in the implementation. The input of the implementation is a Lustre model, and

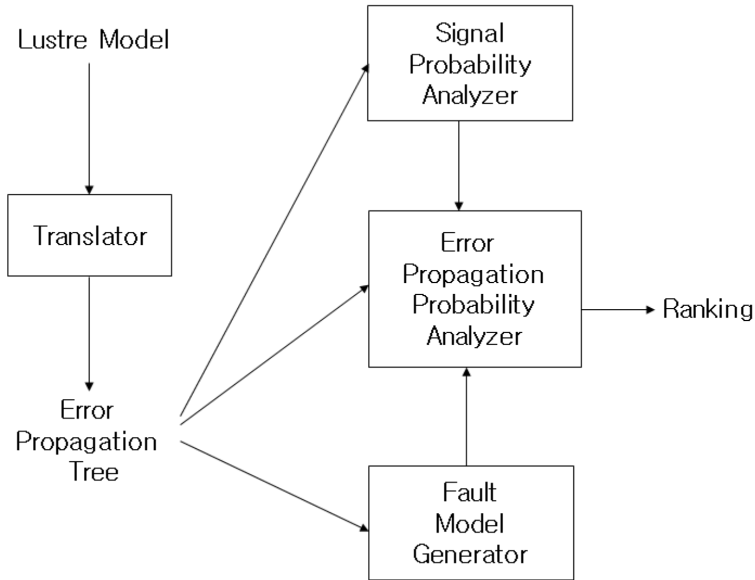


Figure 5.1: System overview

the translator generates an error propagation tree from the Lustre model. The error propagation tree plays the same role as a timed data flow graph during our error propagation analysis, and it is used by other components. The error propagation analyzer synthesizes information from the signal probability analyzer and the fault model generator. It finally generates a ranking of variables of the Lustre model.

5.1.2 Translator

This translator was made by Matt Staats, a member of our research group (Crisys Group at the University of Minnesota). The translator parses the Lustre model to produce the error propagation tree. Technically, it is an error propagation forest since it consists of a set of trees. However, for convenience, we call it the error propagation tree rather than the error propagation graph. The root node of the trees are either output nodes or internal nodes that can be monitored. Internal nodes are made up of

operator nodes, terminal nodes, input nodes, value nodes or variable nodes. Figure 5.2 presents a simple Lustre program and its error propagation tree generated by the translator.

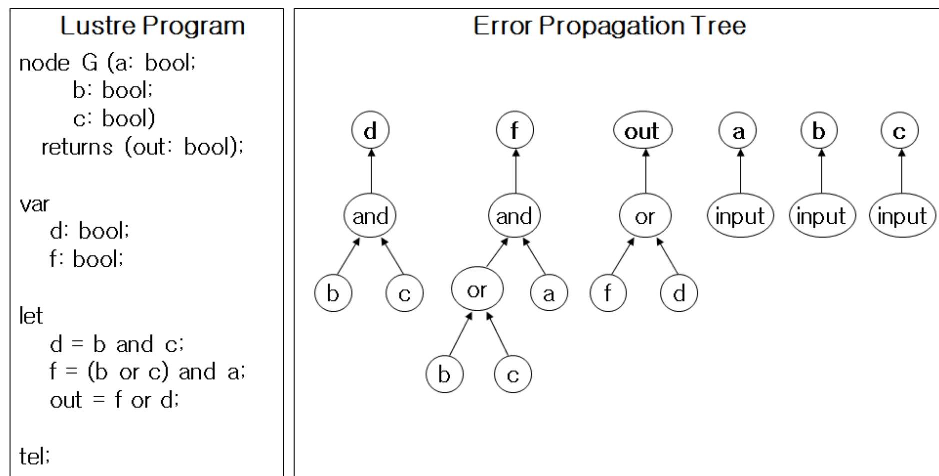


Figure 5.2: A Lustre program and its error propagation trees

5.1.3 Signal Probability Analyzer

The signal probability analyzer determines the signal probability for all boolean domain nodes. To compute the signal probability, it is necessary to predetermine the signal probability for all boolean input nodes. We have two different options to set the signal probability of the boolean input nodes.

Option 1: In this option, we assume the values of boolean input nodes are uniformly distributed. Therefore, we allocate 0.5 for the signal probability of all boolean input nodes.

Option 2: In this option, we can assign a specific number for the signal probability of all boolean input nodes.

5.1.4 Error Propagation Probability Analyzer

Error propagation probability analyzer computes error propagation probability for all nodes in the error propagation tree. This analyzer calls the fault model generator to compute the value of the error propagation attributes $(P_a(n), P_{\bar{a}}(n), P_1(n), P_0(n))$ for the error source node. It also calls the signal probability analyzer to obtain the signal probability of the boolean domain node. With this information, the analyzer generates the error propagation table discussed in Chapter 4. Based on this table, it finally generates a ranking, which is based on the failure probability of the nodes.

5.1.5 Fault Model Generator

Our implementation provides two different fault models, which can assign a specific fault probability for each type of nodes in an error propagation tree.

Fault Model 1

In fault model 1, we assume a uniform distribution of faults over the error propagation tree. Therefore, we allocate an equal fault probability for every node in an error propagation tree. For example, if the number of nodes in the error propagation tree is n, every node will have the fault probability of $\frac{1}{n}$. When a node is selected as a fault node, we assign the error propagation attributes as follows. $P_a(N) = \frac{1}{n}, P_{\bar{a}}(N) = 0, P_1(N) = 0, P_0(N) = 0$

Fault Model 2

In fault model 2, we assign a unique fault probability to each type of node. When we allocate a fault probability to a specific type of node, this probability is divided by the number of the same type nodes in the error propagation tree, and the divided probability will be uniformly distributed to all nodes of the type. For example, if we

assign 0.2 to the logical operator type and the number of logical operator nodes is 2, then each node of a logical operator type has 0.1 probability of being a fault. Thus, the fault probability of each node in fault model 2 depends on the number of nodes with the same type in the error propagation tree.

5.2 Implementation Issues

In our implementation, there are some limitations to precisely computing the rank of nodes in the error propagation tree. Some processes of the error propagation analysis and oracle data selection require very expensive computation which causes the applicability of our implementation to be restricted to small systems. Thus, to increase the applicability of our approach, we compromised the accuracy of the error propagation analysis and oracle data selection to some extent. We introduced two main compromises in our implementation.

5.2.1 Dependence Problem

During the computation of signal probability and error propagation probability, a dependence problem occurs if the input nodes of a logical operator node are dependent on each other. Figure 5.3 shows an example where the dependence problem occurs. The OR operator has two inputs and they are dependent on each other. This means that the value of an input to the OR node is dependent on the value of the other input. In Chapter 3, we introduced a method to deal with the dependence problem when we computed the signal probability and error propagation probability. The basic idea of the method is as follows: we use symbolic computation of the probabilities that is instantiated when the computation is completed.

With this method, we can solve the dependence problem, but the computational cost is exponential to the number of nodes in the worst case [3]. On the other hand,

if we do not consider the dependence problem, the computation time is logarithmic in the number of nodes. To increase the applicability of our implementation, we decided to compromise the preciseness of our computation. Thus, we do not consider the dependence problem in our implementation. However, our pilot experiment shows that the compromise does not seriously affect the final results: the rank of each node.

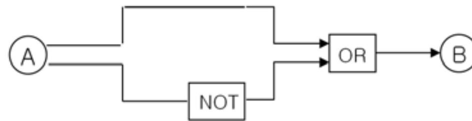


Figure 5.3: Dependence between inputs

5.2.2 Tree Structure vs. Graph Structure

To generate relative ranking, we repeatedly recompute the failure probability of nodes with consideration of the repeated detection of errors. If an error propagation tree is a graph structure, the recomputation process becomes significantly more complicated and time consuming. Furthermore, for some systems, it becomes intractable. However, if the error propagation tree is literally a tree structure, the recomputation time can be significantly reduced. For an example, Figure 5.4 shows both a tree structure and a graph structure of an error propagation tree. Suppose node B has already been selected as an oracle datum. Choosing node C as another oracle datum does not affect the failure probability of node A if the error propagation tree is a tree structure. However, the error propagation tree is a graph structure, so choosing node C as an oracle datum affects the failure probability of node A. As a consequence, every time an oracle datum is chosen, we have to traverse the graph to identify the effect of the oracle datum on the failure probability of other nodes.

To increase the scalability of our implementation, we assume that all error prop-

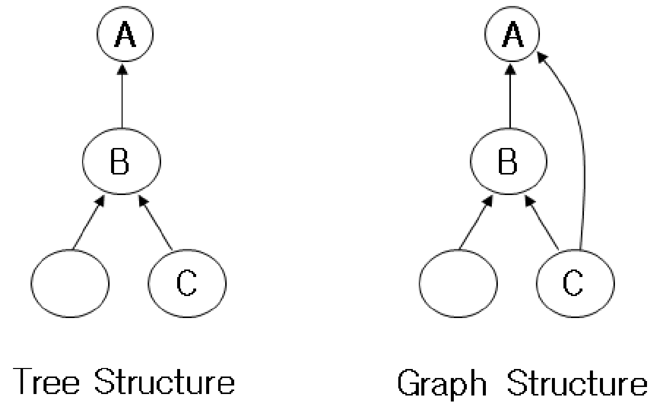


Figure 5.4: The tree and graph structure

agation trees are pure tree structures. Definitely, this simplifying assumption causes an inaccuracy in the failure probability of nodes, and finally it can affect the rank of each node. However, in our pilot experiment, implementation with a pure tree structure significantly reduced the computation time compared to the graph structure, but the inaccuracy was not significant.

Chapter 6

Evaluation

In this chapter, we describe the experiments undertaken to validate our oracle data selection criterion and present the results of the experiments. We first describe the goal of our experiments. We then describe the case examples on which we perform our experiments. This is followed by a detailed description of two different experiments: effectiveness and sensitivity of our oracle data selection criterion. A majority of this chapter is devoted to presenting and discussing the results of these experiments.

6.1 Goal of the Experiments

We have two goals with the experiments; first, to examine the effectiveness of our oracle data selection criterion. Second, to check the sensitivity of our oracle data selection criterion to changes in the assumptions made on, for example, the input signal probabilities. Specifically, we attempt to answer the following questions through the experiments.

1. How effective is the oracle data chosen by our oracle data selection criterion in catching errors in a system?
2. How sensitive is our oracle data selection criterion against variations of signal probability and fault distribution?

For the effectiveness of our oracle data selection criterion, we have two fundamental questions. The first question is about the general effectiveness of our oracle

data selection criterion in fault finding: *"Will the oracle data chosen by our criterion outperform the oracle data chosen by a random approach to oracle data selection?"* If error propagation analysis is a promising approach for oracle data selection, the oracle data chosen by our criterion **should** catch more errors than the oracle data chosen by any naive random approach.

The second question is about the effectiveness of relative ranking over absolute ranking in catching errors. We expect that the oracle data chosen by relative ranking will be more powerful in catching errors than if chosen by absolute ranking. Even though our expectations are based on a mathematical analysis, we are not sure that our analysis in theory will actually have any significant deference in practice. Even if our expectations are correct, we are still interested to determine the level of influence it will demonstrate in finding errors. Therefore, we attempt to investigate this question through experiments

For the sensitivity of our oracle data selection criterion, we investigate the degree of change in a ranking for oracle data selection when the signal probability and fault distribution change. We noted in Chapter 4 that our oracle data selection criterion needs three assumptions about the signal probability, fault models, and test case length to be predetermined. However, in reality there are numerous cases in which the signal probability and fault models can not be determined in advance. As a result, the following question arise: *If our assumptions on the signal probability and fault models used in error propagation analysis are wrong with respect to those used in real testing, how much will it affect the oracle data selection?* Naturally, it would be desirable if the selection approach was not highly sensitive to the assumptions. Nevertheless, should the assumptions have an impact, the subsequent questions arise: *how much will it deteriorate the fault finding capability of the chosen oracle data? Will it be still better than that of randomly selected oracle data?* These questions will be

examined by our experiments in this chapter.

The actual experimental process is described in Section 6.3 and 6.4. The experimental process was performed on four case examples described in the next section.

6.2 Case Examples

In our experiments, we used four case examples from the civil avionics domain. All systems are modeled in the Simulink notation [25]. The Simulink models were automatically translated into a synchronous data flow langauge, Lustre [10] and we used the implementation of Lustre as the basic model in our experiments. The description of the case examples was adopted from [31].

6.2.1 Flight Guidance System

A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired states. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands.

6.2.2 Wheel Brake System (WBS)

The Wheel Brake System (WBS) is a Simulink model derived from the WBS case example found in ARP 4761 [36, 19]. The WBS is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the

aircraft during the taxiing and landing phases, and in the event of a rejected take-off. Braking on the ground is either commanded manually, via brake pedals, or automatically (autobrake) without the need for pedal application. The autobrake function allows the pilot to pre-arm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobrake function will control the brake pressure to provide a smooth and constant deceleration.

6.2.3 Sensor Voting Model

The triplex sensor voter used in our experiment is a slightly simplified version of one developed at Honeywell Laboratories. The design is that of a generic triplex voter: the voter takes inputs from three redundant sensors and synthesizes a single reliable sensor output. Each of the redundant sensors produces both a measured data value and self-check bit (validity flag) indicating whether or not the sensor considers itself to be operational. All valid sensor signals are combined to produce the voter output. If three sensors are available, a weighted average is used in which an outlying sensor value is given less weight than those that are in closer agreement. If only two sensors are available a simple average is used. If only one sensor is available, it becomes the output. There are two mechanisms whereby a faulty sensor may be detected and eliminated: comparison of the redundant sensor signals and monitoring of the validity flags produced by the sensors themselves.

6.2.4 ASW

ASW is a device of an avionics system developed by Dr. Steven P. Miller. This device is responsible for turning on the power to another device when the aircraft drops below a certain altitude. If the altitude cannot be determined for more than two seconds, the ASW indicates a fault. The detection of a fault turns on an indicator

lamp within the cockpit. The DOI is turned back off again if the aircraft ascends above the threshold altitude plus some hysteresis value. The ASW also accepts an inhibit signal that prevents it from turning on power to the DOI or indicating a fault. All other ASW functions are unaffected by the inhibit signal. The ASW also accepts a reset signal that returns it to its initial state.

6.3 Evaluation of Effectiveness

In this section, we describe the experiment we conducted to investigate the effectiveness of our oracle data selection criterion in terms of revealing errors in systems. This section includes the hypotheses for our experiments, experimental process, and experimental results. We first establish the hypotheses for the experiments.

We believe our oracle data selection based on our error propagation analysis should be reasonably effective with respect to fault finding. Thus, our oracle data selection should be more effective than any naive approaches, for example, random selection of oracle data, which does not incorporate any thorough analysis of error propagation. To investigate how effective our oracle data selection criterion is in fault finding, we designed our experiment to test the following hypothesis:

Hypothesis [1] (H_1)**:** The oracle data comprising variables selected through error propagation analysis will provide better fault finding than the oracle data with the same number of variables selected at random.

We also believe that the oracle data chosen through relative ranking will outperform the oracle data chosen through absolute ranking. This belief originates from the notion of *repeated detection* described in Chapter 4. The oracle data from relative ranking are selected in a way to reduce the repeated detections of an error and to

increase the fault finding probability. On the other hand, the oracle data from absolute ranking are generated without considering the repeated detections of an error. Since the repeated detections of an error does not increase the test effectiveness, we believe that oracle data that are chosen to reduce the repeated detections will show the better performance in fault catching. To examine this, we establish the following hypothesis for the experiment.

Hypothesis [2] (H_2): The oracle data chosen through relative ranking will catch more errors than the same number of oracle data chosen through absolute ranking.

6.3.1 Experiment Setup

As mentioned above, the goal in this experiment was to determine the fault finding effectiveness of oracle data selected by our heuristics using error propagation analysis. We performed this assessment on four case examples described in Section 6.2. We first generated mutants of the case examples by seeding a single fault per mutant. We also generated test cases to execute the case examples and mutants together. We then chose oracle data to observe during the execution of the case examples and mutants. Finally, we ran the experiment and recorded the number of faults caught in each oracle data group. It is now possible to determine which oracle data group is more effective with respect to fault finding.

Before going into details of experimental setup, we define a *baseline ranking* as a ranking which is generated by the error propagation analysis with optimal parameters and heuristics. In this case, the optimal parameters and heuristics will be the ones that closely match the way the tests were generated and the faults were seeded to create the mutants. In short, the assumptions used to perform the ranking closely match the conditions faced in the experimental setup. The parameters and heuristics

used are explained in Section 6.3.4. The baseline ranking is expected to show the best performance to catch errors, and this ranking is used as a basis to compare the effectiveness of different oracle heuristics.

For each case example, we performed the following steps.

Step 1, Generate mutants: We randomly generated 250 mutants, each containing a single fault as explained in Section 6.3.2

Step 2, Generate the random test input: We generated a population with 1000 test inputs for each case example in a random manner and then randomly chose the sample test inputs from the population. This is elaborated in Section 6.3.3

Step 3, Generate oracle data: We generated oracle data to investigate each hypothesis.

- Hypothesis [1] : We generated a set of oracle datum with a baseline ranking and 200 sets of oracle datum in a random manner

- Hypothesis [2] : We generated a set of oracle datum with a relative ranking and another set of oracle datum with absolute ranking

Step 4, Run test cases on the mutants and case examples, and monitor the oracle data groups: We ran each mutant and the case example using our test cases, and monitored the oracle data selected in **Step 3**

Step 5, Assess the fault finding capability of each oracle data group: We determined how many mutants were detected by every oracle datum and compared which set of oracle datum is superior in fault finding capability

Step 6, Perform statistical analysis: We determined whether the superiority of an oracle data group in fault finding capability was a statistically significant level or not. The method used for statistical analysis is introduced in Section 6.3.5

6.3.2 Mutant Generation

Mutant generation was performed by Matt Staats, a member of the Crisys Group at the University of Minnesota. The description in this section is adopted from [33].

We created 250 *mutants* (or faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was introduced by either inserting a new operator into the system, or by replacing an operator or variable with a different operator or variable. We seeded the following classes of faults:

Arithmetic: Changes an arithmetic operator (+, -, /, *, mod, exp).

Relational: Changes a relational operator ($=, \neq, <, >, \leq, \geq$).

Boolean: Changes a boolean operator ($\vee, \wedge$, XOR).

Negation: Introduces the boolean $\neg$ operator.

Delay: Introduces the delay operator on a variable reference. (The delay operator causes the implementation to use the stored value of the variable from the previous computational cycle rather than the value computed in the current cycle.)

Constant: Changes a constant expression by adding or subtracting 1 from a int or real constant, or by negating a boolean constant.

Variable Replacement: Substitutes a variable occurring in an equation with another variable of the same type.

Node Call Replacement: Replaces a variable used as a parameter in a node call with another variable of the same type. (This is comparable to changing the parameter used in a function call in a traditional programming language.)

We seed a single fault of a specified fault class using two steps. First, we randomly choose one expression suitable for seeding the specified fault class. Second, we randomly choose a mutation operator (i.e., we determined how to change the expression). We then seed the fault. For instance, to seed an arithmetic fault, we randomly chose one expression from the arithmetic expressions in the model, and then randomly chose a different arithmetic operator to replace the current arithmetic operator. For example, assume we choose the expression '$a + b$'. We then choose to change '+' to either '$-$', '$*$', or '/' and create the mutant accordingly. Note that we did not seed duplicate faults.

In our experiment, we generated mutants so that the *fault ratio* for each fault class was approximately uniform. The term fault ratio refers to the number of mutants generated for a specific fault class versus the total number of mutants possible for that fault class. For example, assume an implementation consists of R Relational operators and B Boolean operators. Thus there are R possible Relational faults and B possible Boolean faults. For a uniform fault ratio, we would seed x relational faults and y boolean faults in the implementation so that $x/R = y/B$.

Note that we did not check that generated mutants were semantically different from the original implementation. This weakness in mutant generation does not affect our results since we are interested in the relative comparisons between oracles, not their absolute fault finding ability.

6.3.3 Test Input Generation

Test input generation was also performed by Matt Staats, a member of the Crisys Group. This description is adopted from [26].

We generated a single set of 1,000 random tests for each case example. Each individual test in these sets contains between 2-10 steps with the number of tests of

each test length distributed evenly in each set of tests. We then generate 100 reduced test suites from the 1,000 random tests. Every reduced test suite for a case example was randomly selected from the 1,000 test sets for that case example.

To control for bias in our results caused by the selection of the test data, we used randomly generated tests of varying lengths instead of tests satisfying a particular coverage criterion. For example, carefully selected, requirements-based, black-box tests are efficient in investigating execution scenarios and are designed to propagate faults to certain outputs [41]. On the other hand, tests generated to satisfy a structural coverage criterion, such as MC/DC, are likely to be very short and there is a possibility that faults encountered will not have the opportunity to propagate to an output [31].

6.3.4 Baseline Ranking Generation

The **baseline ranking** is a ranking of system variables generated by our error propagation analysis, which is expected to show the best performance in fault finding if it is used to choose oracle data. The baseline ranking is generated under the condition that (1) the signal probability of boolean input nodes is 0.5, (2) the error probabilities of nodes in a system are all equal, (3) the test case length is equal to the maximum time delay of the system, (4) relative ranking is used instead of absolute ranking, and (5) an interstate check is used instead of a last-step check. The underlying idea of baseline ranking is (1) to create the parameters of the error propagation analysis corresponding to the condition of our real testing environment, and (2) to use the most promising strategy in oracle data selection. As explained in Section 6.3.3, we generated test inputs in a random manner where we had uniform distribution of boolean domain input between $true$ and $false$. Therefore, we chose the signal probability of 0.5 for the generation of the baseline ranking. Similarly, to create a

mutant, we randomly chose a location to seed a single fault as explained in Section 6.3.2. Thus, we assumed during the generation of the baseline ranking that every node in the system would have an equal probability of being an error node. The test case lengths chosen for our experiments were $2 - 10$, which were far longer than the maximum time delay of our case examples, all having at most $3 - 4$ time delays. Thus, we used the maximum time delay of each case example as the test case length for the generation of the baseline ranking.

For the strategies of oracle data selection, we chose the relative ranking for our baseline ranking instead of the absolute ranking. As explained in Chapter 4, the relative ranking is expected to outperform absolute ranking in fault finding since the relative ranking is optimized to increase the fault finding capability by reducing the repeated detection of a fault while absolute ranking does not consider the repeated detection. Finally, we chose the interstate check for our baseline ranking generation instead of last step check since we expect that the interstate check is more promising to choose oracle data than the last step check.

As a consequence, we believe that the baseline ranking is the most promising heuristic using the error propagation analysis defined in this dissertation to choose oracle data. Thus, we consider the baseline ranking as a reference to indicate the fault finding capability of our oracle data selection criterion.

6.3.5 Chi-square Test

A Chi-square test is an analysis commonly used to test whether distribution of categorical variables of two or more groups differ from one another. The Chi-square test can be used in case the categorical variable is only on count number (i.e., frequency), but not on proportions, means, etc. The Chi-square test is a nonparametric test in that it does not make an assumption about the distribution of scores underlying the

data and only assumes normal distribution of deviations (observed minus expected values). The null hypothesis for testing the independence states whether or not the distributions of the frequency between the two groups differs significantly. The hypothesis is either accepted or rejected after comparison of the Chi-square value to a probability distribution. The Chi-square value is the sum of the squared difference between the expected and observed data which are divided by the expected data in all possible categories.

To use a Chi-square test, a contingency table that is a tabular representation of categorical data is constructed. As an example, a 2×3 contingency table for two groups and three variables is set as in Table 6.1[42]. In this table, n_{ij} denotes the sum of the frequencies for group i and R_i denotes the sum for variable i. N is the sum of all frequencies. After the contingency table is constructed, the Chi-square value is computed using the following formula.

$$X^2 = \sum_{i=1}^{r} \sum_{j=1}^{k} \frac{(n_{ij} - E_{ij})^2}{E_{ij}}$$

where $E_{ij} = \frac{R_i C_j}{N}$, r is number of variables and k is number of groups.

Variable	**Group 1**	**Group 2**	**Combined**
1	n_{11}	n_{12}	R_1
2	n_{21}	n_{22}	R_2
3	n_{31}	n_{32}	R_3
Total	C_1	C_2	N

Table 6.1: An Example of a Contingency Table

Before the Chi-square value can be evaluated, the degrees of freedom for the data set must be determined. Degrees of freedom are the number of independent variables

in the data set. In Table 6.1, the degrees of freedom are calculated as the product of the number of variables minus 1 and the number of groups minus 1. In this example, $(3-1)(2-1) = 2$; thus, there are two degrees of freedom.

Once the degrees of freedom are determined, the value of X^2 is compared with the critical value, $\chi^2_{\alpha,f}$, which can be looked up from the appropriate Chi-square distribution table with a significance level(α) and the degree of freedom(f). For two degrees of freedom and a significance level (α), the critical value associated with $\alpha = 0.05$ for X^2 is 5.99. If the Chi-square value is higher than this critical value, then the null hypothesis $H0$ is rejected.

6.3.6 Experiment Results

In this section, we describe the results of the experiment performed to investigate the effectiveness of our oracle data selection heuristics. To recap, for each case example, we generated 1000 random tests, four rankings (baseline ranking, absolute ranking, interstate check ranking and random ranking), and 250 mutants. We randomly selected test cases from the set of 1000 random tests to generate 100 reduced test suites (their size ranges from 10 to 1000). We then performed data collection by running the 100 test suites against each mutant and case example. We can analyze this data to determine the fault finding ability of the test suites using the oracle data chosen from the four rankings–simply compare the values produced by the case examples against every mutant. The fault finding effectiveness of the test suites and oracle data pair is computed as the number of mutants detected. We perform this analysis for each number of oracle data and test suite for every case example.

To plot our results in graphs, we use build-in function in Matlab called sooth with local regression using weighted linear least squares and a first-degree polynomial model (loess). Note that we smooth the results using the loess method with a span

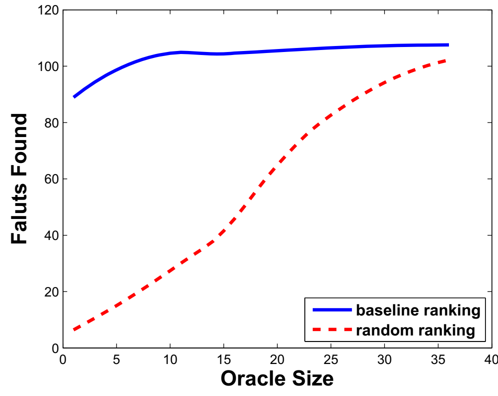
of 70% in this work.

Hypothesis [1], H_1

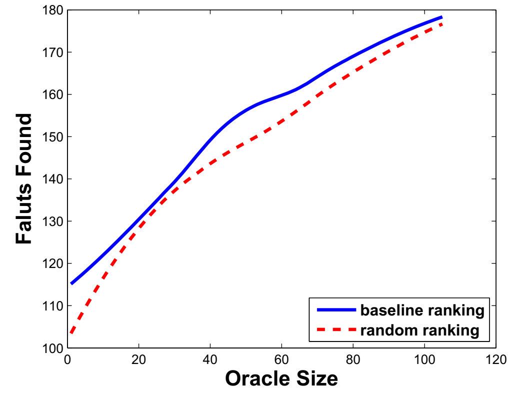
Figure 6.1 shows the number of faults found in the oracle data chosen by the baseline ranking and in a random manner in the four systems: FGS,WBS, Sensor Voter, and ASW. In the figure, the x-axis is the oracle data size, which is equivalent to the number of monitored variables in the system, and the y-axis shows the number of faults found. Note that the number of fault found in oracle data is an averaged value of the faults that are caught with 100 different test suites and 200 random oracle data set.

Generally, the oracle data chosen by our oracle data selection heuristics outperform the oracle data chosen in a random manner as seen in Figure 6.1. Specifically, in the FGS system, our heuristic dramatically improves performance as the oracle size increases. Even with one selected oracle datum using the baseline ranking technique, it identifies around 70 faults on average. With 2 selected oracle data, it identifies, on average, 101 faults, which is 78% of the maximum possible fault detections using these test suites. In contrast, random selection shows a linear relationship between oracle data size and the number of faults found. Even with 34 oracle data, random selection identifies less than 100 faults, on average, which could be identified only with 2 oracles with our heuristic.

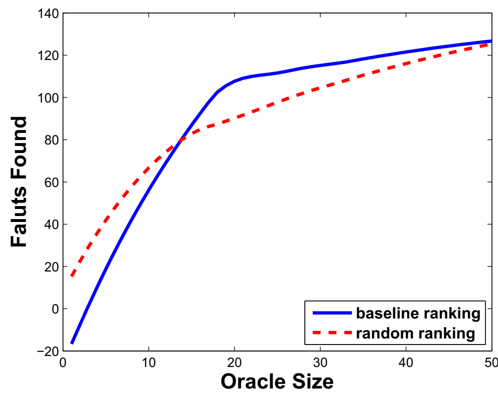
The WBS and ASW system show a slightly different result than the FGS result. The gaps between our heuristic and random selection are not as significant as FGS. As can be seen in Figure 6.1(b) and Figure 6.1(d), however, our heuristic outperforms random selection over oracle size to the maximum 38% (for ASW) and 33% (for WBS), indicating that our heuristic is beneficial in identifying many more faults with the same number of oracles.



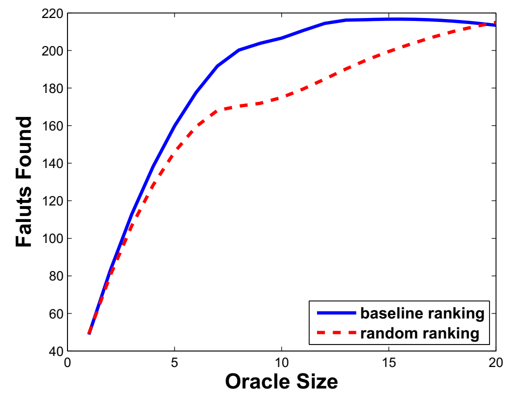
(a) FGS System



(b) WBS System



(c) Sensor Voter System



(d) ASW System

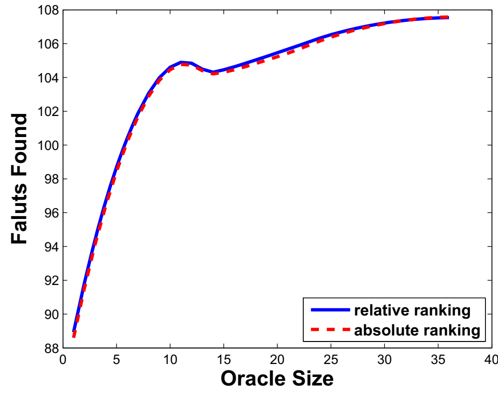
Figure 6.1: Effectiveness of oracle data: baseline ranking vs. random manner

However, the Sensor Voter system shows a different tendency in fault finding capability compared to other systems, results that fall short of our expectation. Until an average of 10 oracle data, the oracle data chosen by our oracle data selection heuristic underperform the oracle data chosen by a random manner. This result was somewhat disappointing since we believed that our oracle data would always be more effective than the random ones. Through an investigation of the internal structure of the Sensor Voter system, we noticed that the Sensor Voter has several unusual constructs in the system. This may be the reason that the Lustre program is mechanically translated from the Simulink Model. Table 6.2 shows an example of the construct in the Sensor Voter system.

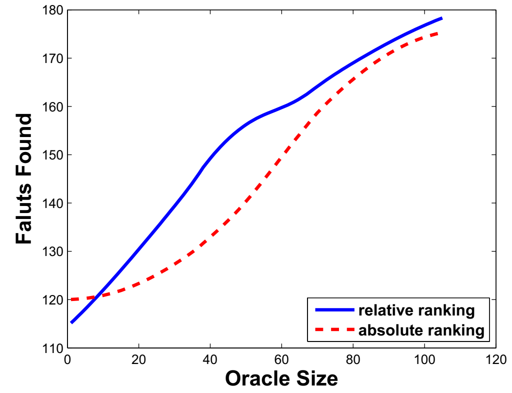
$Node1 \ = \ Node2 \ \times \ 0$

Table 6.2: An example expression of the Sensor Voter system

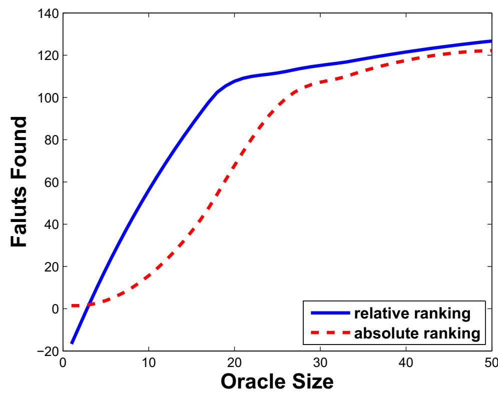
The expression, "$Node2 \times 0$" is a very unusual expression since it does not make any sense from a programming point of view. For this construct, no errors on $Node2$ can be propagated to $Node1$. Thus, there is no reason to monitor $Node1$ in real testing. However, in our heuristics, the failure probability of $Node2$ is all transferred to Node1 using the formula in Table 3.7. Thus, $Node1$ is a promising candidate for oracle data in our heuristic while its fault finding capability in real testing is non-existent. This is the reason that our oracle data underperformed the oracle data in random selection until the number of oracle data became 10. As the number of oracle data exceeded 10, our oracle data showed better performance in fault finding than the oracle data in a random manner. The maximum difference in fault finding between our oracle data and random oracle data was 20% for the Sensor Voter system.



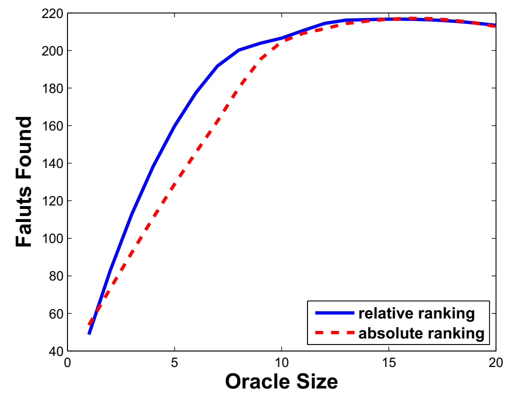
(a) FGS System



(b) WBS System



(c) Sensor Voter System



(d) ASW System

Figure 6.2: Effectiveness of oracle data: relative ranking vs. absolute ranking

Hypothesis [2], H_2

Figure 6.2 shows the number of faults detected in the oracle data chosen by relative ranking and absolute ranking in four systems. Except for the FGS system, the oracle data chosen by relative ranking outperformed the oracle data chosen by relative ranking. In these systems, the number of fault finding of the relative ranking sharply increased as the number of oracle data increased compared to absolute ranking. The maximum difference in fault finding between relative ranking and absolute ranking is 15%(WBS), 43%(ASW) and 75%(Sensor Voter).

In contrast, as can seen in Figure 6.2(a), the FGS system does not show any difference in fault finding capability between the oracle data of relative ranking and those of absolute ranking. In fact, the oracle data chosen by relative ranking outperformed the oracle data from absolute ranking, showing at most 2% of the maximum difference in fault finding. This difference is almost negligible. If there is no difference in fault finding between the two different sets of oracle data, one reason may be that the set of oracle data is identical. However, our investigation of the two rankings (relative and absolute ranking) of the FGS system demonstrated that these two rankings are somewhat different. Through the investigation of fault finding capability of each variable, we noticed that 94% of the errors were revealed by only two variables, and the error catching rate from other variables was significantly low. The rank of the two variables in relative and absolute ranking was identical. That is the reason that the fault finding capability of the two rankings was almost equal. Furthermore, since the two variables are not computationally connected to each other, there were no repeated detections of errors between them. Thus, their ranks in the relative ranking and absolute ranking did not change.

Statistical Analysis

In this section, we statistically analyze the results in Figure 6.1 and 6.2 to determine if the hypotheses, H_1 and H_2, stated previously in the introduction of this chapter are supported.

To evaluate H_1 and H_2, we formulate our respective null hypotheses $H0_1$ and $H0_2$ as follows:

$H0_1$: Oracle data chosen by baseline ranking will find the same number of faults as oracle data chosen in a random manner.

$H0_2$: Oracle data chosen by relative ranking will find the same number of faults as oracle data chosen by absolute ranking.

To accept H_1 and H_2, we must first reject $H0_1$ and $H0_2$, respectively. Rejecting $H0_1$ and $H0_2$ implies that the number of faults caught by the oracle data chosen from each of the baseline and relative rankings come from different populations which are random ranking and absolute ranking, respectively. In other words, this implies that either the oracle data chosen from baseline ranking and relative ranking have more fault finding than the oracle data chosen by random or absolute ranking, respectively, or vice versa.

To accept H_1 and H_2, after rejecting $H0_1$ and $H0_2$, we examine the number of faults found from each oracle data and determine if the oracle data chosen from baseline and relative ranking have more fault finding than the oracle data chosen by a random manner and absolute ranking, respectively. If so, we accept H_1 and H_2.

Since we have already shown that the oracle data chosen by baseline ranking and relative ranking outperform the oracle data chosen by a random manner and absolute ranking, respectively, we just need to reject $H0_1$ and $H0_2$ to accept H_1 and H_2, respectively.

Observations of our experiments are drawn from an unknown distribution. Thus, we applied the Chi-square test, which is free from an assumption of the distribution, to evaluate the hypotheses. The details of the Chi-square are described in Section 6.3.5.

Table 6.3 lists the Chi-square value and critical value for the null hypotheses and states whether the corresponding null hypotheses are rejected for each case example. We reject the null hypotheses ($H0_1$ and $H0_2$) if $X^2 > \chi^2_{\alpha,f}$, where f is the number of degrees of freedom; in our case examples, it is 1000 since the number of the group is 2 and the number of the sample size is 1001). α is the upper percentage point of the Chi-square distribution; we adopt its level of 0.05 as the significant level.

As seen, $H0_1$ is rejected with a significance level $\alpha = 0.05$ for all case examples. As a result, H_1 is supported for all case examples. Except the FGS and WBS system, $H0_2$ is also rejected with a significance level $\alpha = 0.05$. Thus, we can accept H_2 except the FGS and WBS system.

Case Examples	$H0_1$			$H0_2$		
	X^2	$\chi^2_{0.05,1000}$	Result	X^2	$\chi^2_{0.05,1000}$	Result
FGS	11340	1075	rejected	1.39	1075	not rejected
WBS	3076	1075	rejected	162	1075	not rejected
Sensor Voter	15834	1075	rejected	5555	1075	rejected
ASW	2116	1075	rejected	1728	1075	rejected

Table 6.3: Hypotheses Evaluation

6.4 Evaluation of Sensitivity

In this section, we evaluate the sensitivity of our oracle data selection criterion in the face of variations of signal probability and fault distribution. Recall that our oracle data selection criterion requires that the signal probability for boolean input and fault distribution should be predetermined to perform an error propagation analysis. *If the given signal probability and fault distribution for error propagation analysis turn out to be wrong, how much will it affect the selection of the oracle data and how will it affect the fault finding capability of the chosen oracle data?*.

As expected, our oracle data selection criterion will not be sensitive to variations in the signal probability of boolean input nodes. The signal probability of boolean input nodes only affects the failure probability of the boolean domain node of a system. Hence, the failure probability of non-logical nodes in a system will not be affected by the variation of signal probability. As a result, we expect that the variation of failure probability of each node in a system will be limited to a small number of nodes. Thus, we anticipate that our oracle data selection criterion will not be sensitive to the variation of signal probability.

In contrast, we expect that our oracle data selection criterion will be sensitive to variations of the fault model. Since the rank of a node is completely determined by the failure probability of the node and the failure probability is determined by the fault model, the variation of the fault model will cause the rank of nodes to change. As a result, we expect that our oracle data selection criterion will be sensitive to the fault model

To validate our expectation on the sensitivity of our oracle data selection criterion, we perform experiments using the same case examples with experiments for the effectiveness evaluation. Before going into detail about the experiments, we first define some methods to determine the sensitivity of our oracle data selection criterion

in the following section.

6.4.1 Rank Distance

To examine the sensitivity of our oracle data selection criterion, we use *rank distance* [30] which indicates the difference between the two rankings. If a ranking significantly changes in response to a change in one of the parameters to the ranking algorithm, the oracle data that are chosen by the changed ranking will also be significantly different from the one chosen by the previous ranking. We can therefore determine the sensitivity to the variation of the value of the parameter by examining if the oracle data chosen by each of the two rankings are different. If a ranking is not significantly disturbed in response to the changes of a parameter value, our oracle data selection criterion is insensitive to the parameter selection.

The rank distance is defined as follows. In the following definition, we assume that all rankings consist of exactly the same objects.

Definition 32 ***Rank distance:***
Let α_1, α_2 be the two rankings of a set of n objects. ***Rank distance*** *between α_1 and α_2 is*

$$\sum_{i=1}^{n} |\alpha_1(i) - \alpha_2(i)|$$

where $\alpha_1(i)$ represents the i'th object in the α_1.

Definition 33 ***Maximum rank distance:***
The maximum rank distance of a ranking α_1 is the rank distance between α_1 and α'_1, where α'_1 is reversely sorted from α_1.

For example, if α_1 is $< a, b, c, d >$, α'_1 becomes $< d, c, b, a >$.

Definition 34 ***Normalized rank distance:***
Let α_1, α_2 *be the two rankings of a set of* n *objects. The* ***normalized rank distance*** *between* α_1 *and* α_2 *is computed as the rank distance between* α_1 *and* α_2 *divided by the maximum rank distance of* α_1 *or* α_2*.*

The normalized rank distance can be an indication on how different the two rankings are. If the two rankings, α_1 and α_2, are identical, the normalized rank distance between them is 0; whereas, if they are reversely lined up, the normalized rank distance is 1.

To establish a yardstick for the sensitivity of our oracle data selection criterion in terms of the rank distance, we devised a measure by which it can be determined whether a rank distance is within the limit or not.

Definition 35 ***Base Rank Distance:***
The ***base rank distance*** *of a ranking,* α_1*, is the maximum possible rank distance between the two rankings,*α_1 *and* α'_1*, where* α'_1 *is generated from* α_1 *and every object in* α_1 *changed its rank within* 10% *of the number of objects in* α_1*.*

Definition 36 ***Normalized Base Rank Distance:***
The ***normalized base rank distance*** *is computed as the base rank distance of a ranking divided by the maximum rank distance of the ranking.*

Table 6.4 shows an example of computing the base rank distance of α_1. Since the number of objects in α_1 is 10, every object can move at most one object distance to make α'_1. Thus, every two adjacent objects in α_1 is switched in α'_1.

The base rank distance can be used as an indication of how far objects of a ranking are relocated from the original ranking. If a ranking has a greater rank distance than the base rank distance of original ranking, the rank of every object in

Base Rank Distance(α_a)							
α_a	a	b	c	d	e	f	base rank distance :
α'_a	b	a	d	c	f	e	6

Table 6.4: Base Rank Distance

the ranking should change, on average, over 10% of the number of objects from the original ranking. We applied this base rank distance as a yardstick in determining the sensitivity of our oracle data selection criterion, although it was not based on the theoretical or empirical evidence.

6.4.2 Critical Variable Variation

A limitation of the normalized rank distance as a measure of sensitivity of our oracle data selection criterion is that it can not distinguish a rank change of an object with a higher rank from a rank change of an object with a lower rank. For example, in a ranking $\alpha_1 =< a, b, c, d, e >$, the normalized rank distances are the same in both cases where a and b are exchanged and d and e are exchanged. In our experiment for effectiveness evaluation, we have noticed that few variables(mostly, located in higher positions in rankings) have a more critical effect in catching errors than other variables that all showed very similar but poor capability in catching errors. As a consequence, with respect to fault finding capability, we have to differentiate the rank changes between objects with high ranks from the rank changes between objects with low ranks. As mentioned above, the normalized rank distance is not a suitable measure to check the amount of sensitivity in terms of fault finding capability. Therefore, the normalized rank distance should be restricted as an indication of the amount of variation between two rankings.

To compensate for the limitation of the normalized rank distance, which is unable

to measure the variation of fault finding capability caused by the variation of ranking, we designed another variation measure, *critical variable variation*. In our pilot experiments, we noticed that failure probability of individual variables in a system varies significantly, so for some systems under investigation, there were a few variables that had substantially more failure probabilities than other variables. Recall that only two variables in the FGS system caught 94% of the errors in the system. We call those variables *critical variables*. To formally define these critical variables, we used the failure probability of variables rather than the actual fault finding capability of variables, which is often unobtainable. The critical variables are defined as the variables whose failure probabilities are more than two standard deviations from the mean. Subsequently, we define **critical variable variation** as follows.

Definition 37 ***Critical variable variation:***
Let $T(\alpha)$ be the set of variables which are critical variables in a ranking α and α_1, α_2 be the two rankings with the same objects. The ***critical variable variation*** *from α_1 to α_2 is*

$$\frac{|T(\alpha_1) - T(\alpha_2)|}{|T(\alpha_1)|}$$

If the critical variables of α_1 are totally different from the critical variables of α_2, the critical variable variation between them is 1, and if the critical variables of the two rankings are identical, the critical variable variation is 0. The critical variable variation can be used as an indication of how many critical variables have changed between the two rankings. Consequently, the critical variable variation can provide insight about the sensitivity of our oracle data selection criterion in terms of fault finding capability.

6.4.3 Experiment Setup

The goal in this experiment was to determine the sensitivity of the oracle data selection criterion over the signal probability and fault models. To perform this experiment, we first generated the baseline rankings for the four case examples. We then computed the normalized base rank distance of the baseline rankings. We also generated the rankings with the skewed signal probability and with the skewed fault model for the four case examples. We then computed the normalized rank distance and critical variable variation between the baseline rankings and the skewed rankings (the rankings with the skewed signal probability and fault models). Following is a detailed description of the experimental process.

Step 1, Generate the baseline ranking: We generated the baseline rankings for each case example as described in Section 6.3.4. Note that the baseline rankings were generated with the parameters (1) the signal probability, 0.5, (2) uniform distribution of faults over all nodes in a system, and (3) a test case length that is equivalent to the maximum time delay of a system.

Step 2, Compute the normalized base rank distance: We computed the normalized base rank distances from the baseline rankings generated in Step 1.

Step 3, Generate rankings with the skewed signal probability: We generated the skewed signal probability rankings for each case example as follows.

- Low signal probability: To check the effect of a low signal probability on our oracle selection criterion, we assigned a signal probability of 0.2 to all boolean input nodes.

- High signal probability: To check the effect of a high signal probability on our oracle selection criterion, we assigned a signal probability of 0.8 to all boolean input nodes.

Step 4, Generate rankings using the skewed fault models: We generated the skewed fault model rankings for each case example as follows.

- ID fault model: In this fault model, only ID nodes could have a fault.
- Logical operator fault model: In this fault model, we assumed only logical operator nodes could have a fault.

Step 5, Compute a normalized rank distance: We computed the normalized rank distances between the baseline rankings and skewed rankings for each case example.

Step 6, Compute a critical variable variation: We computed the critical variable variation between the baseline rankings and skewed rankings for each case example.

6.4.4 Experiment Results

In this section, we describe the results of the experiment performed to investigate the sensitivity of our oracle data selection heuristics.

Sensitivity to Signal Probability

For each case example, we generated three rankings (α_b, α_2 and α_8) under the conditions of the test case length, 3 (test case length is 4 for the Sensor Voter system) and the uniform fault distribution. The test case length was chosen because the maximum time delay by Pre operators for the case examples was 3 (for the Sensor Voter system, 4). α_b is the baseline rankings with the signal probability, 0.5 for all the boolean input nodes. Similarly, α_2 is the skewed rankings with the signal probability, 0.2 and α_8 is the skewed rankings with the signal probability, 0.8. Table 6.5 shows the sensitivity of our oracle data selection criterion against the variation of signal probability.

Sensitivity to Signal Probability				
System	Ranking	Normalized Rank Distance	Normalized Base Rank Distance	Critical Variable Variation
FGS System (# of variables: 76)	α_2	0.111	0.117	0
	α_8	0.117		0
WBS System (# of variables: 110)	α_2	0.09	0.2	0
	α_8	0.175		0
Sensor Voter System (# of variables: 56)	α_2	0.031	0.17	0
	α_8	0.031		0
ASW System (# of variables: 30)	α_2	0.104	0.2	0.5
	α_8	0.102		0

Table 6.5: Sensitivity to signal probability

Overall, as expected, our oracle data selection criterion is generally not sensitive to the variations of signal probability. For every case example, the normalized rank distance is less than the normalized base rank distance, meaning that the average rank variation of each object is less than 10% of the number of objects in a ranking. The result also shows that the critical variable variation is insensitive to the variations of signal probability, indicating that the fault finding capability of the skewed rankings will not significantly change compared to the baseline ranking.

As a result, we expect that even though we do not know the input distribution of a system under test in advance, our oracle data selection criterion will still be helpful to choose the most promising oracle data.

Sensitivity to Fault Model

For each case example, we generated three rankings (α_b, α_{ID}, and α_{LO}) with the test case length, 3 (test case length is 4 for the Sensor Voter system) and the signal probability, 0.5 for all the boolean input nodes. The α_b ranking is the baseline ranking under the condition of uniform fault distribution. This means that all nodes in the system have the same error probability. For the ranking α_{ID}, only ID nodes can have a fault and for the ranking α_{LO}, only logical operator nodes can have a fault. Table 6.6 shows the sensitivity of our oracle data selection criterion to the fault models. In the table, the rankings, α_{ID} and α_{LO} are compared with the baseline ranking, α_b.

Sensitivity to the Fault Model				
System	Ranking	Normalized Rank Distance	Normalized Base Rank Distance	Critical Variable Variation
FGS System (# of variables: 76)	α_{ID}	0.745	0.17	1
	α_{LO}	0.601		0.25
WBS System (# of variables: 110)	α_{ID}	0.242	0.2	0
	α_{LO}	0.649		0.83
Sensor Voter System (# of variables: 56)	α_{ID}	0.26	0.17	0
	α_{LO}	0.578		1
ASW System (# of variables: 30)	α_{ID}	0.024	0.2	0.5
	α_{LO}	0.628		1

Table 6.6: Sensitivity to Fault Model

As expected, there is a high degree of sensitivity of our oracle data selection criterion to fault models. For most cases, the normalized rank distances are far greater than the normalized base rank distance. In addition, the critical variable variation

is also considerably great, implying that the fault finding capability of oracle data chosen by our criterion is sensitive to the fault models.

This result can be explained as follows. If the computation of a node does not depend on either logical operator nodes or ID nodes, the node will have a *zero* failure probability in our fault models. Similarly, if a node is computationally linked to many logical operator nodes or ID nodes, it will have a high failure probability. Thus, the failure probability of a node in our fault models strongly depends on the number of logical operators and ID nodes in its computation. Consequently, the ranking based on these skewed fault models can significantly deviate from the baseline ranking which assumes a uniform distribution of a fault over all system nodes, which subsequently causes our oracle data selection criterion to be sensitive to these fault models.

6.5 Discussion

Here we elaborate on our findings about the effectiveness and sensitivity of our oracle data selection criterion and discuss their implication.

6.5.1 Effectiveness of Our Oracle Data Selection Criterion

The effectiveness of our oracle data selection criterion in selecting oracle data has been demonstrated in our experiment. The oracle data chosen using our oracle data selection criterion show significantly better fault finding capability than the oracle data chosen randomly at the significant level $\alpha = 0.05$. This indicates that, given two groups of oracle data with the same size (one group from our heuristic, the other group selected at random), the oracle data of our group is likely to outperform the other group in fault finding. Furthermore, the analysis also revealed that the oracle data chosen by relative ranking tend to show equal or better fault finding capability compared to the oracle data chosen by absolute ranking. Therefore, unless testers

are willing to monitor all variables (output and internal variables), our heuristics (particularly, relative ranking) will provide the guidelines for choosing the monitoring variables, which will show relatively good performance with respect to error detection.

While our oracle data selection heuristics is generally more effective than a random approach, we noticed that the effectiveness of our oracle data selection criterion significantly varies depending on the system. Our criterion shows impressive effectiveness when it is applied to predict the fault finding capability of nodes for the FGS system, while it does not show the same amount of effectiveness for the WBS system. A closer investigation revealed a positive correlation between the efficiency of our oracle data selection criterion and the existence of *critical variables* which heavily influence the fault finding ability of oracle data. If a system has the critical variables (i.e., most nodes have poor fault finding capability, whereas a small number of nodes have significantly better fault finding probability), our oracle data selection criterion can show considerable effectiveness in fault finding compared to the random selection as shown in the FGS case example. In the FGS system, only $2-3$ nodes contributed to catching all errors, while most nodes did not catch any errors. On the other hand, if a system does not have critical variables(i.e., most variables have similar capability in fault finding), our oracle data selection does not show any impressive effectiveness in predicting the fault finding capability of nodes even though it outperforms the randomly selected nodes. The WBS system is a representative system that does not have such critical variables since around half of the nodes in the WBS system have very similar and good fault finding capability. As seen in Figure 6.1(b), the effectiveness in fault finding in WBS is not as good as shown in FGS.

6.5.2 Accuracy of Our Oracle Data Selection Criterion for identifying Critical Variables

As mentioned above, critical variables of a system play a critical role in determining the effectiveness of our oracle data selection criterion. We believe that our oracle data selection criterion has a considerably level of effectiveness in identifying the critical variables in a system if they exist. To check the effectiveness of our oracle data selection criterion to choose the critical variables as oracle data, we examined the fault finding ability of each individual variable for every case example. This examination revealed that the fault finding ability of individual variables significantly varies depending on systems. For example, over 50% of the variables of the FGS system reveal, on average, less than 1 fault, whereas more than 90% of the variables of the WBS system reveal, on average, at least 1 fault. Despite this variability among systems, for every system, there exists a few variables that reveal substantially more faults than the average variables. In our case examples, few variables were identified as critical variables(FGS, WBS, Sensor Voter and ASW system have $2, 5, 3$, and 1 critical variables, respectively). These critical variables reveal 3 to 335 times more faults than the other variables; for instance, the average fault finding of noncritical variables of the FGS system is 0.2, while the fault finding of a critical variable of FGS is 67. Clearly, the inclusion or exclusion of these critical variables in oracle data has a significant impact on the effectiveness of the oracle data. Thus, we investigated the capability of our oracle data selection criterion to identify the critical variables. As seen in Table 6.7, we investigated the rank of each critical variable in the baseline ranking and identified its relative position in the ranking.

As seen in the table, the results are very promising. Except for the Sensor Voter system, most critical variables are positioned within 20% in the baseline ranking. Recall that the rank of nodes in the Sensor Voter system is not precisely reflecting

System	Critical Variable(CV)	Rank in Baseline Ranking	Relative Position in Ranking
FGS (# of variables: 76)	CV1	1	1%
	CV2	2	3%
WBS (# of variables: 110)	CV1	1	1%
	CV2	2	2%
	CV3	3	3%
	CV4	5	5%
	CV5	88	80%
Sensor Voter (# of variables: 56)	CV1	11	20%
	CV2	13	23%
	CV3	17	30%
ASW (# of variables:30)	CV1	5	17%

Table 6.7: The rank of critical variables in baseline ranking

the fault finding capability of the nodes since the unusual expressions (i.e., $Node1 = Node2 \times 0$) made the error propagation analysis incorrect. Without the expressions, we expect that the critical variables in the Sensor Voter system will have higher ranks than this result. Nevertheless, all critical variables of Sensor Voter system are located in the upper 30% of the baseline ranking.

Note that the rank of CV5 in the WBS system is significantly deviated from the ranks of other critical variables. The rank of CV5 is 88'th of 110 nodes. When we check the rank of CV5 in the absolute ranking(which does not consider repeated detections of errors), the rank slightly move forward to 61. Recall that around half

of variables in the WBS system have very similar and good fault finding capability. Hence, the rank of CV5 in the baseline ranking does not necessarily mean the inability of our oracle data selection criterion in identifying the critical variables.

To sum it up, the result in Table 6.7 demonstrates the capability of our oracle data selection criterion to identify the critical variables in a system.

6.5.3 Sensitivity of Our Oracle Data Selection Criterion

According to our experiments, our oracle data selection criterion is not sensitive to the variation of signal probability and is sensitive against the fault models. Although we use a relatively strict yardstick (the normalized base rank distance) to determine the sensitivity, the variation of signal probability does not indicate any noticeable changes in ranking. In contrast, fault models show considerable effects on the failure probability of nodes and thereby cause significant changes of the rank of nodes. This result suggests that a tester who uses our heuristics needs to spend more time anticipating the programmer's tendency of making mistakes rather than the boolean input distribution of test cases.

During our experiment, we also noted that the sensitivity of our oracle data selection criterion also depended upon system characteristics. For example, the number of specific logical operator nodes can affect the sensitivity of our oracle data selection criterion against signal probability. If the signal probability of input nodes decreases (close to 0), the masking probability of AND nodes increases, and accordingly, it causes the failure probability of the connected nodes to decrease. On the other hand, if the signal probability of the input nodes increases, the masking probability of the OR nodes increases. As an extreme case, if a system does not have any boolean inputs, the ranking of this system will not be affected by signal probability. Similarly, if the main function of a system is arithmetic operations, the error in the

logical operator nodes does not have a significant effect on the ranking variation. For these reasons, system characteristics play a critical role in the sensitivity of our oracle data selection criterion. Therefore we should consider the system characteristics to anticipate the amount of sensitivity of our criterion.

6.5.4 Effect of Ranking Variation on Fault Finding Capability of Oracle Data

As mentioned in the discussion above, our oracle data selection criterion is relatively sensitive for fault models. Thus, an incorrect assumption on fault distribution will generate a skewed ranking that is deviated far from an ideal ranking with the correct fault distribution. Then, a question arises. *If a ranking deviates from the baseline ranking, how will it affect the fault finding capability of the oracle data chosen by the ranking?* If the fault finding capability of oracle data is not sensitive to the ranking variation, we may need to be less concerned about the sensitivity of our oracle data selection criterion. On the other hand, if the fault finding capability turns out to be sensitive to the ranking variation, we should pay more attention to reducing the rank variation. To investigate the impact of ranking variations on the fault finding capability of oracle data, we examined the fault finding capability of skewed rankings that are generated for sensitivity evaluation and compared it with the fault finding capability of baseline ranking and random ranking.

Figure 6.3 - 6.6 shows the experimental results of the fault finding capability of each ranking.

In the FGS system, we did not observe any significant sensitivity in fault finding capability. Moreover, the most deviated ranking (α_{LO}) shows much better performance in fault finding than the random ranking. As mentioned before, FGS has two critical variables that caught most of the errors in the system. Thus, the critical

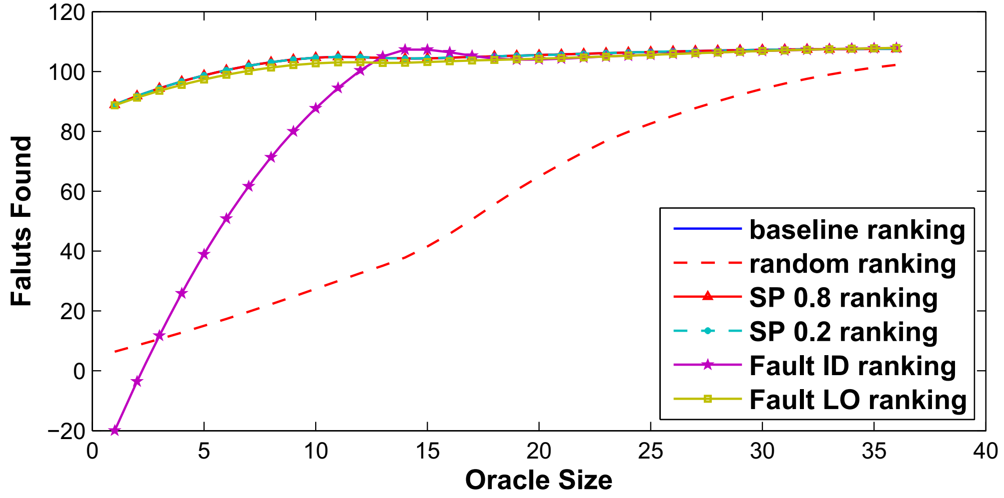


Figure 6.3: Fault finding capability of skewed rankings(FGS system)

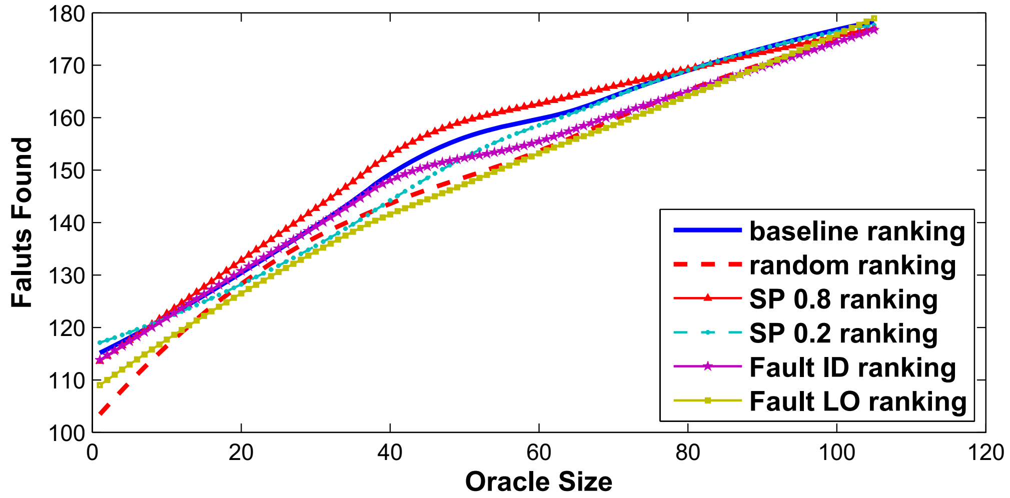


Figure 6.4: Fault finding capability of skewed rankings(WBS system)

variable variation has more impact on fault finding than the variation of the whole ranking. Note that the fault finding capability of α_{LO} is not good at the top rank nodes because the critical variable variation of α_{LO} is 1.

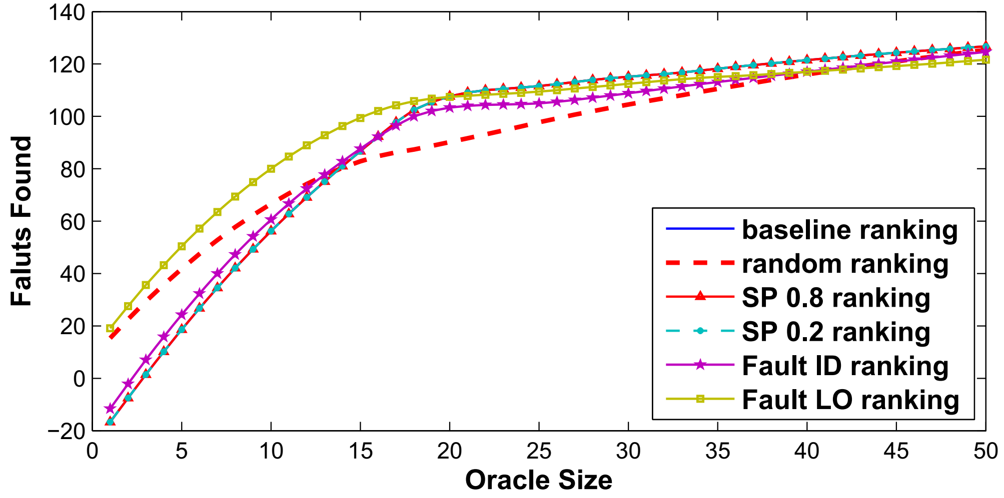


Figure 6.5: Fault finding capability of skewed rankings(Sensor Voter system)

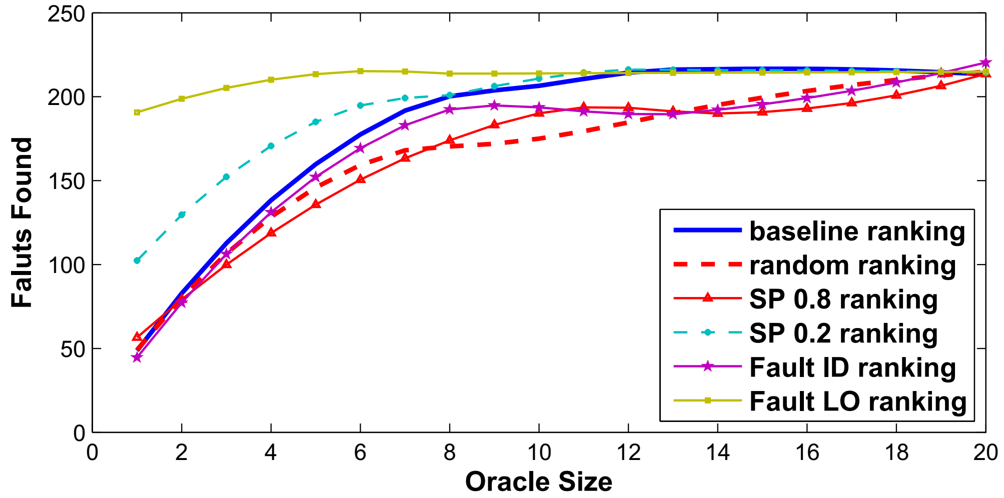


Figure 6.6: Fault finding capability of skewed rankings(ASW system)

In the WBS, the ranking α_{LO} shows less fault finding capability than random ranking, because this ranking has significant variations in the whole ranking (normalized rank distance: 0.649) and critical variables (critical variable variation: 0.83).

In the Sensor Voter system, most rankings show equal or better fault finding capability compared to baseline ranking. Interestingly, α_{LO} has a much better fault finding capability than baseline ranking even though its ranking variation(normalized rank distance:0.578 and critical variable variation: 1) is very high. As noted, the baseline ranking of the Sensor Voter system became inaccurate by the expression, "$Node1 = Node2 \times 0$". However, in α_{LO}, every node is arranged by the failure probability of the logical operator node since only the logical operator node can have a fault. Thus, the expression, "$Node1 = Node2 \times 0$", which is an arithmetic operation, did not affect the ranking of α_{LO}. As a result, α_{LO} can reflect the fault finding capability of nodes better than baseline ranking.

For the ASW system, α_{LO} has a much better fault finding capability than baseline ranking even though its ranking variation(normalized rank distance:0.628 and critical variable variation: 1) is very high . Through our investigation, we concluded that the performance was obtained by chance. There are only four nodes in the ASW system that are connected to logical operator nodes. Therefore, these nodes are located in a high position in the ranking, α_{LO}. Among these nodes, there is a critical variable by chance that can lead to a significant improvement in the fault finding capability of α_{LO}.

Overall, we cannot identify a strong correlation between the whole ranking variation (normalized rank distance) and fault finding capability. However, for most cases, we found that the critical variable variation can be used as an indication of how much the fault finding capability of a ranking has deviated from that of baseline ranking.

6.6 Threat to Validity

While our results are promising, we can identify several threats to the validity of our experiments.

First, the results of our experiments are derived from a small set of examples, which poses a threat to the generalization of the results. Nevertheless, we believe that the examples we used in our experiment are representative, and our results are generalizable to systems within the same domain.

Second, the execution probability of each code segment of a program is not considered in our analysis. Although we assume uniform input distribution, the execution probability of specific parts of a program may not be uniform. The execution probability is more dependent on the control structure of the program. If the execution probability for every code segment in a program is not uniform, our error propagation analysis may not precisely predict the real fault finding capability of variables.

Finally, we assumed in our approach that the system under test contains a single error. Thus, a single error is seeded into each mutant in our experiments. A single error per mutant, however, is unrealistic in real systems. A real system may have more than one error which may lead to a coincidental recovery of the system from errors. However, a coincidental recovery is not taken into account in our error propagation analysis. If coincidental recovery is frequent, the accuracy of the error propagation analysis will be compromised.

Chapter 7

Conclusion and Future Research Directions

In this dissertation, a criterion of oracle data selection was proposed so that we can maximize the fault finding capabilities of oracle data. In particular, our error propagation analysis plays a crucial role in choosing oracle data. The analysis techniques are based on the masking and propagation probability when an error is propagated through other nodes. The timed data flow graph was also introduced to provide the infrastructure for error propagation analysis. The notion of failure probability was defined as a quantitative measure of the error revealing capabilities of a node. Based on the failure probability, four approaches are suggested to rank all nodes in terms of error revealing capabilities: absolute ranking, relative ranking, interstate check and last step check. The absolute ranking simply lines up all nodes according to the failure probability of each node. In contrast, the relative ranking line up all nodes in a way to maximize the error revealing capability of oracle data as well as to minimize the repeated detection of an error from several different nodes. The interstate check lines up all nodes according to the failure probability over every execution step while the last step check lines up all nodes based on the failure probability of the last step.

Our approach is implemented in order to demonstrate the feasibility of our error propagation analysis and is evaluated with respect to effectiveness and sensitivity.

Our experimental results show that there is a strong positive correlation between our oracle data selection criterion and fault finding effectiveness. In addition, fault finding capability of the oracle data chosen by our criterion significantly outperforms

randomly selected oracle data. These promising results imply that we achieved our goal to design an oracle data selection criterion to improve the test effectiveness. However, some parts need to be further investigated so as to determine the most potential benefits of our approach. The emerging research issues regarding this dissertation are as follows.

- Implementation of our oracle data selection technique can be improved in terms of accuracy and efficiency. The status quo cannot deal fairly with the dependency problem among incoming edges of an node. In the worst case, exponential time complexity is required to resolve the dependency problem although the worst case, in general, does not frequently occur. Thus, a more efficient algorithm may deal more precisely with this dependency problem.

- We used random test cases for the evaluation of our oracle data selection criterion. In reality, there are several techniques to generate test cases which force an error in a system to propagate through output nodes. Thus, an intelligent design of test cases will have an effect on the fault finding capability of nodes. In future research, we want to investigate the effect of our oracle data selection criterion with respect to these intelligently designed test cases.

- Currently, our oracle data selection criterion focuses more on the effectiveness of fault finding. That is, if any node contains a higher potential for error detection than the others, it is selected as an oracle datum. The error propagation analysis, however, can also be used to produce another criterion of oracle data selection, namely, the hardness-based oracle data selection criterion. This criterion focuses on finding the most difficult errors. The oracle data selected by this criterion can show better performance for finding errors which is very difficult errors to be caught without directly monitoring the oracle data.

- Our error propagation analysis mainly focuses on domain specific language: synchronous data flow language. An extension of the application domain of our error propagation analysis could be another future research issue.

In sum, our error propagation analysis is a very promising technique to apply to various domains. We hope this dissertation continues to inspire new ideas of software testing, debugging, and maintenance in software engineering research.

Bibliography

[1] W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, B. Yu, and A. Mili. Error propagation in software architectures. In *IEEE METRICS*, pages 384–393. IEEE Computer Society, 2004.

[2] L. Andeerson. *Fault tolerance, principles and practice*. Prentice Hall, 1981.

[3] G. Asadi and M. B. Tahoori. An analytical approach for soft error rate estimation in digital circuits. In *ISCAS (3)*, pages 2991–2994. IEEE, 2005.

[4] H. Asadi, M. B. Tahoori, and C. Tirumurti. Estimating error propagation probabilities with bounded variances. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007), 26-28 September 2007, Rome, Italy*, pages 41–49. IEEE Computer Society, 2007.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering (FOSE'07) at 29th International Conference on Software Engineering*, pages 85–103, 2007.

[7] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[8] R. Black. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. Wiley, 2007.

[9] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Software Eng*, 30(11), 2004.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM Symp. Principles Program. Lang. (POPL)*, pages 178 – 188, Munich, Germany, 1987.

[11] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In T. Ostrand and E. Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 171–181, 1993.

[12] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Klower Academic Press, 1993.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[14] R. G. Hamlet. Software quality, software process, and software testing. *Advances in Computers*, 41, 1995.

[15] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 23 July 2001.

[16] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS)*, pages 161–172, Goteborg, Sweden, July 2001. IEEE Computer Society.

[17] W. E. Howden. A functional approach to program testing and analysis. *IEEE Transactions On Software Engineering*, SE-12, October 1986.

[18] J.C. Laprie. Dependability: Basic concepts and associated terminology. Technical Report 31, PDCS, May 1990.

[19] A. Joshi and M. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122–135. Springer-Verlag, Sept 2005.

[20] C. Kaner. A course in black box software testing.

[21] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software.* Wiley, 1999.

[22] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-maC: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2), 2004.

[23] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec*, 4(1-2), 2005.

[24] Machado. Testing from structured algebraic specifications. In *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, 2000.

[25] Mathworks Inc. Simulink Product Web Site. Via the world-wide-web: http://www.mathworks.com/products/simulink.

[26] C. Muñoz, editor. *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, number NASA/CP-2010-216215, NASA Langley Research Center, Hampton VA 23681-2199, USA, April 2010.

[27] G. Myers. *The Art of Software Testing*. Wiley, 2004.

[28] K. P. Parker and E. J. McCluskey. Probabilistic treatment of general combinational networks. *IEEE Trans. Computers*, 24(6):668–670, 1975.

[29] R. Patton. *Software Testing*. SAMS, 2006.

[30] M. Popescu and L. P. Dinu. Rank distance as a stylistic similarity. In *Coling 2008: Companion volume: Posters*, pages 91–94, Manchester, UK, August 2008. Coling 2008 Organizing Committee.

[31] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 161–170. ACM, 2008.

[32] A. Rajan, M. W. Whalen, M. Staats, W. Deng, and M. P. Heimdahl. The effect of program and model structure on the fault finding ability of mc/dc test suites. In *In Proceedings of International Symposium on Software Testing and Analysis (ISSTA), Submitted 2008. Available at http://crisys.cs.umn.edu/ISSTA08.pdf.*, 2008.

[33] A. Rajan, M. W. Whalen, M. Staats, and M. P. E. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In S. Liu, T. S. E. Maibaum, and K. Araki, editors, *ICFEM*, volume 5256 of *Lecture Notes in Computer Science*, pages 86–104. Springer, 2008.

[34] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *ICSE*, pages 105–118, 1992.

[35] D. S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.

[36] *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[37] S. Sankar and R. Hayes. ADL - an interface definition language for specifying and testing software. In *Workshop on Interface Definition Languages*, pages 13–21, 1994.

[38] The Free Software Foundation. The GNU NANA homepage.

[39] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, Aug. 1992.

[40] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, Nov. 1982.

[41] M. Whalen, A. Rajan, and M. Heimdahl. Coverage Metrics for Requirements-Based Testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.

[42] C. Wohlin, P. Runeson, M. Hösrt, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[43] Q. Xie and A. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 2006. To appear.

Appendix A

Algorithm for Relative Ranking

The computation of relative ranking starts from the error propagation table. For the first member of the relative ranking, the node with the highest failure probability is chosen. For the second member, we recompute the failure probability of all nodes in the error propagation table with consideration of the probability of repeated detections of errors with the chosen node in the ranking. This process is repeated until every node is located in the ranking. We explain this process in detail using the example of Figure A.1. We first define the error propagation probability through an edge in the timed data flow graph.

Definition 38 ***Error propagation probability through an edge:***
$PE(A, B, C)$ is the error propagation probability from an error source node A to C through an edge, (B, C)

In Figure A.1, $PE(A, A, D)$ represents the probability that the error of node A propagates to node D through the edge, e_1. The computation of $PE(A, A, D)$ can be performed by the analysis of the error propagation attributes of node D. For example, suppose node D is an AND operator. Then, the error propagation attributes, $P_a(D)$ are defined as $P_a(D) = P_a(B)P_a(A) + P_1(B)P_a(A) + P_a(B)P_1(A)$ using the formula in Table 3.15. In this expression, the error propagation probability through edge e_1 is computed by deleting all terms that include $P_a(B)$ and $P_{\bar{a}}(B)$ since they represent the error propagation probability through node B (i.e., through edge e_2 and e_5).

The term, $P_1(B)P_a(A)$ is the error propagation probability through edge e_1. Thus, $PE(A, A, D) = P_1(B)P_a(A)$.

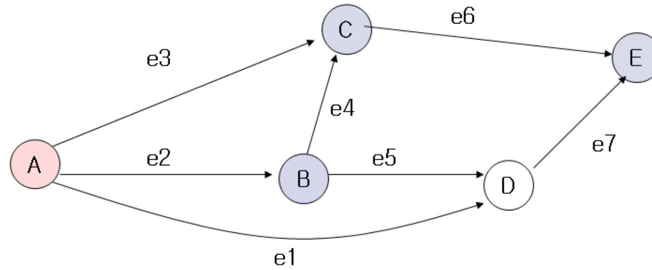


Figure A.1: Recompute the error propagation probability

In Figure A.1, suppose node A is the error source node and node B, C, and E are chosen as oracle data. We want to recompute the error propagation probability from A to D. To recompute the error propagation probability of node D with the given oracle data, we consider the benefit(avoiding repeated detection with the oracle datum) to monitor node D. If the error of node A propagates to one of the oracle data, there is no benefit to monitoring node D to catch the error of node A. Therefore, the benefit of monitoring node D occurs when the error of node A is masked out to all oracle data and it propagates to node D. In Figure A.1, the benefit of monitoring node D to catch the error in node A occurs when the error of node A is masked out in the edges e_2, e_3 and e_6, and the error propagates through e_1. Thus, the error propagation probability from A to D is computed as follows. $PP(A, D) = (1 - PE(A, A, C))(1 - PE(A, A, B))(1 - PE(A, D, E))PE(A, A, D)$

Every time a new oracle datum is selected, we need to reconfigure the error propagation tree by deleting some connections among oracle data. If an error source node is determined during the error propagation analysis, some edges of the error propagation tree need to be removed to further perform the error propagation analysis. For example, in Figure A.1, edges $e4$ and $e6$ have to be deleted in the graph since the

error of node A can not propagate through the edges. The reason is that some oracle data(node B and C) have already caught the error of node A. After deleting the unnecessary edges among oracle data, we further process the recomputation of the failure probability of each node. Algorithm 7 shows how to remove an edge using the error propagation attributes of a node. Recall that the error propagation attributes of a node have all the information about the path through which an error propagates from an error source node. This algorithm is used in Algorithm 8, which is the main algorithm to recompute the error propagation probability for every node.

Algorithm 7 Remove the connection between the two nodes u and v

1: **if** node u is an ancestor node of node v **then**

2: Unfold the error propagation attribute of v to u. Remove all terms containing $P_a(u)$ and $P_{\bar{a}}(u)$.

3: **else**

4: Unfold the error propagation attribute of u to v. Remove all terms containing $P_a(v)$ and $P_{\bar{a}}(v)$.

5: **end if**

Algorithm 8 Recompute the error propagation probability

1: Let OD be a set of oracle data and u be a new member of OD

2: Let $OD_{ancestor}(v) = \{u_1 \mid u_1 \in OD \textit{ and } u_1 \in ANCESTOR(v)\}$

3: Let $OD_{descendent}(v) = \{u_1 \mid u_1 \in OD \textit{ and } u_1 \in DESCENDENT(v)\}$

4: Run algorithm 7 for u and v where $v \in OD$.

5: **for** each row w of error propagation table(M) **do**

6: Set $M(w, u) \leftarrow 0$.

7: **for** each cell x in the column where $M(w, x) \neq 0$ **do**

8: Run algorithm 7 for v and x where $v \in OD_{ancestor}(x)$.

9: Let $Mask_{descendent} = \prod_{v \in OD_{descendent}(x)}(1 - PP(x, v))$

10: Let $Mask_{ancestor} = \prod_{v \in OD_{ancestor}(x)}(1 - PP(w, v))$

11: Set $M(w, x) = Mask_{descendent} * Mask_{ancestor} * PP(w, x)$

12: **end for**

13: **end for**