ABSTRACT

Title: A Study of Software Input Failure Propagation Mechanisms

Yuan Wei, Doctor of Philosophy, 2006

Directed By: Associate Professor Carol Smidts
Department of Mechanical Engineering

Probabilistic Risk Assessment (PRA) is a well-established technique to assess the probability of failure or success of a system. Classical PRA does not consider the contributions of software to risk. Dr. B. Li and C. Smidts have established a framework to integrate software into PRA which recognizes the existence of four classes of risk contributors: functional, input, output and support failures. Input/Output failures have been shown to make up 57.4 % of the failures experienced during software development of major aerospace systems and have been at the origin of a number of major accidents such as the Mars Polar Lander. This research quantifies the contribution of the input failures. More specifically, this dissertation 1) defines the concept of input failure, 2) studies the related propagation mechanisms, 2) estimates the propagation probability for different types of input failures, and 3) applies the fault propagation analysis to the framework of integrating software into PRA.

The dissertation defines the concept of artifact as a reference point to identify expected inputs and consequently input failures (inputs which differ from the expected ones). Input failures are divided into value-related failures (including value, range, type and amount failures) and time-related failures (including time, rate and duration failures). Value failures are examined first. The concept of masking areas and flat parts is defined, and the dissertation proposes an Image Reconstruction Method (IRM) to estimate the propagation probability of input value failures. This method is proven to require less number of test cases than one that could be based on random testing to reach the same relative error.

For the other input failure modes, the dissertation reveals how they transform to the data state error and formalizes their propagation criteria so that the IRM can be applied to estimate the propagation probability.

The contributions are thus:

1. Clear definition of the concept of input failure;
2. Definition of a systematic process of identification and quantification of the contributions of input failures to risk;
3. Systematic analysis of the propagation mechanisms of each type of input failures.

A STUDY OF SOFTWARE INPUT FAILURE PROPAGATION MECHANISMS

By

YUAN WEI

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:
Associate Professor Carol Smidts, Chair
Professor Marvin L. Roush
Professor Mohammad Modarres
Associate Professor Joseph Bernstein
Associate Professor Dave Akin

© Copyright by
Yuan Wei
2006

Acknowledgements

I wish to express my sincere gratitude to Dr. Carol Smidts, for without her immense help in guiding my research, this dissertation would have been impossible. As an advisor, she assisted me in every aspect from research brainstorming to writing this dissertation.

I owe special thanks to the contributions of Dr. Manuel Rodriguez for his tremendous help in the study of the propagation of time-related failure modes presented in this dissertation.

I would also like to thank Dr. Ming Li for his constructive suggestions at the beginning of the research.

I am fortunate to have been able to work on this project with a talented and dedicated team of UMD researchers consisting of Dr. Ming Li, Dr. Manuel Rodriguez, Dr. Dongfeng Zhu, Dr. Bin Li, Dr. Avik Sinha, Wende Kong, Ying Shi, Sushmita Ghose, Anand Ladda and Jun Dai. Special thanks are presented to them for their help and the support they provided to this project.

Finally, I would like to thank Dr. Marvin Roush, Dr. Mohammad Modorres, Dr. Joseph Bernstein, and Dr. Dave Akin for agreeing to be on my committee and Dr. Mosleh for not minding being taken off the committee.

I would also like to express my sincere gratitude to the NASA Office of Safety and Mission Assurance for supporting this work through the NASA SARP program managed by the NASA IV&V facility.

Table of Contents

Preface...Error! Bookmark not defined.
Acknowledgements ii
Table of Contents iii
List of Tables vii
List of Figures viii
1 Introduction 1
1.1 Research Objective 1
1.2 Statement of the Problem 1
1.3 Approach 4
1.4 Dissertation Organization 5
1.5 Contributions 6
2 Review of Fault Propagation Analysis 8
2.1 Importance of Fault Propagation 8
2.2 Fault Propagation Analysis 9
2.2.1 Homogeneous Propagation 10
2.2.2 RELAY Model 11
2.2.3 PIE Model 15
2.2.3.1 Execution Probability 15
2.2.3.2 Infection Probability 15
2.2.3.3 Propagation Probability 16
2.2.4 Error Permeability 16
2.2.5 Fault Propagation in Software Architecture 17
3 Input Failure and Artifact 19
3.1 Definitions 19
3.2 Examples of expected input and input failure 24
3.3 Construction of the Artifact 25
3.4 Construction of the Operational Profile 28
3.4.1 Generation of the expected input I_e 29
3.4.2 Generation of an Operational Profile for Input Failures 31
3.4.2.1 Generating $\mathcal{OP}_{hw}$ 31
3.4.2.2 Generating $\mathcal{OP}_{sw}$ 33
3.4.2.3 Generating $\mathcal{OP}_{human}$ 34
3.4.2.4 Generating $\mathcal{OP}_e$ and $\mathcal{OP}_f$ from $\mathcal{OP}_{hw}$, $\mathcal{OP}_{sw}$, and $\mathcal{OP}_{human}$ 34

3.5 Types of Input Failures 35
4 Fault Propagation of Incorrect Input Value 37
4.1 Data State and Data State Error 37
4.2 Information Flow Transfer 38
4.2.1 Data Dependent Transfer 38
4.2.2 Control Dependent Transfer 39
4.3 Propagation Behavior 39
4.3.1 Error Flow Model 39
4.3.2 Cancellation Mechanisms 40
4.3.3 Major Contributors of Masking 41
4.3.4 Multi-layer Traps 44
4.3.5 Potential Mask Area and Potential Propagate Area 45
4.3.6 Single Fault and Multiple Faults 48
4.4 Propagation Probability Estimation 49
4.4.1 Probability Representation and System Overview 49
4.4.2 Image Reconstruction Method 52
4.4.2.1 Data Generation 53
4.4.2.2 Image Reconstruction 53
4.5 Discussion of Image Reconstruction Method 54
4.5.1 Dependency Analysis 54
4.5.2 Error Analysis 58
4.5.3 Efficiency Analysis 60
4.5.4 Scalability Study 63
4.5.5 Application for the different locations in Software 64
5 Fault Propagation of other Value-Related Failure Modes 67
5.1 Type Failure 67
5.1.1 Data Type 67
5.1.2 Type Checking 69
5.1.2.1 Coding stage 70

5.1.2.2 Compilation stage ... 71
5.1.2.3 Executing stage ... 73
5.1.3 Type Conversion ... 74

5.1.4 Summary ... 82

5.2 Range Failure ... 84
5.2.1 Definition ... 84

5.2.2 Out of Range Failure ... 85

5.3 Amount Failure ... 88
5.3.1 Definition ... 88

5.3.2 Propagation of Amount Failure ... 89

6 Propagation of Time-Related Failure Modes ... 93
6.1 Time-Related Failure Modes ... 93
6.2 Timing Failure ... 93
6.2.1 Definition ... 93

6.2.2 Computational Model ... 94

6.2.3 Too Late (and Omitted) Failure Mode ... 96

6.2.4 Too Early Failure Mode ... 101

6.3 Rate Failure ... 102
6.3.1 Definition ... 102

6.3.2 Floors and Ceilings ... 103

6.3.3 Propagation Criteria ... 103

6.3.4 Too Fast Rate Failure ... 104

6.3.4.1 Sporadic Mode ... 104
6.3.4.2 Passive Mode ... 105
6.3.4.3 Active Mode ... 110
6.3.5 Too Slow Failure Mode ... 121

6.3.5.1 Passive Mode ... 122
6.3.5.2 Active Mode ... 123
6.3.5.3 Polling System ... 124
6.3.6 Summary ... 125

6.4 Duration Failure 126
6.4.1 Definition 126

6.4.2 Too short failure & too long failure 126

6.4.3 Role of the Duration 127

6.4.3.1 Too Short Duration Failure 128
6.4.3.2 Too Long Duration Failure 129
7 Application 133
7.1 PACS 133
7.2 Application process 135
8 Conclusion and Future Research 148
8.1 Advantages of fault propagation analysis 148
8.2 Limitation of the Methods and Future Research 149
Appendice A 151
Bibliography 161

List of Tables

Table 3-1 Example Security File 31
Table 4-1 PMA-PPA Matrix 47
Table 4-2 Experiment Results 61
Table 4-3 N/$N_{testing}$ for different npp* and k (r=0.1, z=1.96) 62
Table 4-4 Number of interpolation points required for different npp*, k, r=0.1 64
Table 4-5 Calculation time (s) for different r, npp*, k, r=0.1 (l=128k) 64
Table 5-1 Storage of Data Type 75
Table 5-2 Type Conversion Operations 76
Table 5-3ASCII Code 78
Table 5-4 Results for different conversions 82
Table 6-1 Example for Too Fast Failure (1) 112
Table 6-2 Example for Too Fast Failure (2) 114
Table 6-3 Summary for the propagation of too slow failure 125
Table 6-4 Summary for the propagation of too fast rate failure 126
Table 7-1 Characteristics of SwipeCard 137
Table 7-2 Applicability of the Failure Modes 138
Table 7-3 Result of Fault Propagation Analysis on "SwipeCard" 143
Table 7-4 Operation Profile for SwipeCard 143
Table A-1 Description of Object Toolbar 153
Table A-2 SRS Statements Indication 154
Table A-3 Major Fields Description in a Transition 156
Table A-4 Run Settings Window Options 159

List of Figures

Figure 2-1 Example Module 12
Figure 2-2 Single Chain 13
Figure 2-3 Multi-Chain 14
Figure 3-1 Input to a Software Component in a System 19
Figure 3-2 Location where input failure occurs 20
Figure 3-3 Input Domain with Perfect SyRS 22
Figure 3-4 Input Domain with Complete and Incorrect SyRS 22
Figure 3-5 Input Domain with Incomplete and Correct SyRS 22
Figure 3-6 Typical Input Domain 23
Figure 3-7 Software and System Time Windows 24
Figure 3-8 Example Scenario 30
Figure 3-9 Generating $\mathcal{OP}_{hw}$ 33
Figure 3-10 Generating $\mathcal{OP}_{sw}$ 34
Figure 3-11 Input Failure Modes 36
Figure 4-1 Data Dependent Transfer 38
Figure 4-2 Control Dependent Transfer 39
Figure 4-3 Error Flow Model 40
Figure 4-4 Propagation Behavior 41
Figure 4-5 Multi-Layer Trap 45
Figure 4-6 PMA&PPA 47
Figure 4-7 System Overview 50
Figure 4-8 P_{ti} and P_{wi} Calculation 51
Figure 4-9 Image Reconstruction Method 52
Figure 4-10 Flat Parts Identification 53
Figure 4-11 Algorithm of Flat Parts Identification for single fault 54
Figure 4-12 Independent Range distributions 55
Figure 4-13 Dependent Range distributions 56
Figure 4-14 Dependent Range Distribution (special case) 56
Figure 4-15 Error Analysis 58
Figure 4-16 Example System 60
Figure 4-17 Non-propagation Analysis for a Fault located in the Software 65
Figure 4-18 Mapping in the Head Function 66
Figure 5-1 Transform form incorrect type failure to incorrect value failure 67
Figure 5-2 Type checking in different stages 70
Figure 5-3 Example of Type Checking (Visual Basic) 71
Figure 5-4 Statically vs. Dynamically Typed Variable 72
Figure 5-5 Representation of Integer 76
Figure 5-6 Floating Point Number Layout 76
Figure 5-7 Result of Conversion 82
Figure 5-8 Propagation of Incorrect Type 84
Figure 5-9 Variable and its address 85

Figure 5-10 Propagation of Out of Range Failure 88
Figure 5-11 Propagation of Too Little Amount Failure.. 90
Figure 5-12 Propagation of too much amount failure... 92
Figure 6-1 Time Window for Input 94
Figure 6-2 Computational Model for timing failure 95
Figure 6-3 Simplified Algorithm 98
Figure 6-4 Algorithm to Calculate Probability for predefined data......................... 100
Figure 6-5 Propagation of Delayed Failure Mode 100
Figure 6-6 Too Fast and Too Slow Failure Modes 103
Figure 6-7 Input and Output Series.. 104
Figure 6-8 Interruption System 105
Figure 6-9 Dropped Inputs in Too Fast Rate Failure... 108
Figure 6-10 Too Fast Rate Failure in an Event Driven System (v_e=2, v_f=3, n=1).... 109
Figure 6-11 Too Fast Input Rate causes Too Fast Output Rate.............................. 110
Figure 6-12 Buffer Behavior 1 (v_e=1; v_f=2; n=2) 111
Figure 6-13 Example for Too Fast Rate Failure (2) 113
Figure 6-14 Example for Active Mode (Buffer Behavior 1) 115
Figure 6-15 Popped Inputs in the Too Fast Rate Failure 117
Figure 6-16 Example for Active Mode (Buffer Behavior 2) 118
Figure 6-17 Polling System 119
Figure 6-18 Simplified Polling System 120
Figure 6-19 Too Slow Failure... 121
Figure 6-20 Time Stretch 122
Figure 6-21 Too Slow Failure in the Temperature Monitoring System 123
Figure 6-22 Too Slow Failure in Polling System 125
Figure 6-23 Duration Failure 127
Figure 6-24 Propagation of the “Too Short” Duration Failure 129
Figure 6-25 Input with “Too Long” Duration is Treated as Several Inputs 130
Figure 6-26 Propagation Illustration of Redundant Inputs 131
Figure 6-27 Propagation of the “Too Long” Duration Failure 132
Figure 7-1 Application Process.. 135
Figure 7-2 ESD for Exit System 137
Figure 7-3 Propagation of "too late" failure 140
Figure 7-4 Propagation of "too early" failure 140
Figure 7-5 Propagation of timing failures in SwipeCard... 140
Figure 7-6 Propagation of the "too short" duration failure 141
Figure 7-7 Propagation of the "too little" Failure 141
Figure 7-8 Propagation of the “too little” amount failure in SwipeCard................. 142
Figure 7-9 General propagation behavior of the "too much" amount failure 142
Figure 7-10 Propagation of the "too much" amount failure in SwipeCard.............. 143
Figure 7-11 TestMaster Model - Mainframe 144
Figure 7-12 TestMaster - Amount Failure 145
Figure 7-13 TestMaster Model – SwipeCard... 145
Figure 7-14 TestMaster Model - Value Failure 145
Figure A-1 TestMaster Project Window.. 151
Figure A-2 Entry Level (Root) Model... 152

Figure A-3 Create Objects in TestMaster 152
Figure A-4 Adding Transitions 155
Figure A-5 Edit Transition Window 155
Figure A-6 Edit New Variable 157
Figure A-7 Transition Edited 158
Figure A-8 Run Setting 159

1 Introduction

1.1 *Research Objective*

The objective of this dissertation is to define the concept of software input failures, study the propagation mechanisms for software input failures, and to estimate the propagation probabilities for different types of software input failures. Also, this dissertation tries to apply the fault propagation analysis to the framework of integrating software into probabilistic risk assessment (PRA).

1.2 *Statement of the Problem*

Probabilistic Risk Assessment (PRA) is a methodology consisting of techniques to assess the probability of failure or success of a system. In many modern technological systems, especially safety critical systems such as space systems, nuclear power plants, medical devices, defense systems, etc., PRA has been proven to be a systematic, logical, and comprehensive methodology for risk assessment. However, unfortunately, classical PRA practice ignores the contributions of software due to a lack of understanding of the software failure phenomena. This is in conflict with the fact that the software is playing an increasingly important role in modern systems. Hence, we are trying to develop a methodology to account for the impact of software on system failure that can be used in the classical PRA analysis process. PRA usually answers the following four questions:

- What can go wrong? That is what are the initiators or initiating events that lead to adverse consequence?
- What are the consequences of things going wrong?
- How likely are these undesirable consequences? Or what are the probabilities of the consequences?
- How confident are we about our answers to the above questions?

PRA has been applied in many modern systems, especially in safety critical systems. Unfortunately, the classical PRA ignores the contributions of software to risk. To account for it, a framework of integrating software into PRA was introduced by B. Li and C. Smidts [1-3]. In the framework, Li and Smidts introduce a software-related failure modes taxonomy, in which the failures are categorized into input failures, output failures, functional failures and support failures. According to a validation of this failure modes taxonomy which was performed by NASA Johnson Space Center (JSC) [4], the functional failures and I/O failures are the two major failure modes. Input failures often occur in modern systems. One famous example is the Mars Polar Lander (MPL) failure. The Polar Lander was the first attempt to land on Mars since the Mars Pathfinder mission of 1997. The lander and microprobes were in excellent health during launch and the nine-month transit to Mars. On December 3, 1999, about ten minutes before it was expected to land on the south polar region of Mars, the lander lost contact with Earth and it was never regained. The premature shutdown of the descent engine is the most likely cause for the failure of the mission. The three landing legs sent spurious signals to the MPL's computer convincing it the legs had touched down on the Martian surface and thus turned off the descent engine

used to slow the spacecraft down in the final seconds before landing. This case can be classified as an interaction failure and more specifically an input failure, probably an incorrect value.

In their framework, Li and Smidts also proposed a testing-based approach to quantify the probabilities of end states in the event sequence diagrams (ESD). Their method is valid. However, their approach just quantifies the contribution of functional failures while other failure modes have not been taken into account. Also, just as its name implies, it is based on testing and the result is statistical. If an analytical or semi-analytical method could be applied, the time required for the quantification may be reduced. To do that, one should show how a fault residing in a software component is masked or cancelled.

The understanding of fault propagation is an important field of study for the software community. Faults remaining in software may or may not cause the software to fail. Sometimes, software generates correct outputs even if the input provided to it is incorrect.

Fault propagation is a complex process and the propagation probability, the probability that a fault arising in a location of the software propagates to the software output, is difficult to estimate. However, understanding how the fault propagates and its propagation probability is very important in probabilistic risk assessment because the probability that the software fails due to a fault can be expressed as the product of the probability that the fault is executed and its propagation probability. The reliability of a software component can be expressed as:

$$R_{SW} = 1 - \sum_{i=1}^{N} p(FM_i)\,pp(FM_i)$$

Where

R_{SW} is the reliability of the software component under study,

$p(FM_i)$ is the probability that the ith failure mode occurs,

$pp(FM_i)$ is the probability that the ith failure mode propagates to the output,

N is the number of failure modes which can occur in the software component.

Fault propagation is a very complex process. Different failure modes may have different propagation behaviors. The research focuses on the propagation mechanisms for different types of software input failures.

1.3 Approach

There are many types of software failures and different researchers have proposed different software failures taxonomy [1, 3, 5-10]. In this dissertation, the taxonomy proposed by Bin Li and Carol Smidts is used. According to their taxonomy, the input failures can be categorized into two groups of failure modes: value-related and time-related failures. Value-related failure modes cover the characteristics such as value, type, range, and amount. Time-related failure modes cover the characteristics such as time, duration, and rate. Different input failures may have their own special propagation behaviors. In this dissertation, the propagation mechanisms for different software input failures are described. To reach the objective of this dissertation, the following steps are performed in order:

1. Study the propagation of incorrect input values. It is reasonable to select the incorrect value as the first target of study, because the incorrect value directly causes an erroneous data state that is the

medium to propagate the fault. This step studies the propagation of the data state error and how this error vanishes during propagation. Following the analysis, a method to estimate the propagation probability of incorrect value failures is proposed.

2. Study the propagation of other value-related input failures (incorrect input range failure, incorrect input type failure, and incorrect input amount failure). More specifically, this step analyzes 1) how the propagation of those input failures transforms to the propagation of an incorrect input value or a data state error, or 2) how those input failures are detected or cancelled before they are transformed into a data state error.

3. Study the propagation of time-related input failures. Time-related failure modes cover characteristics such as input time, rate, and duration. In this step, a computational model for the timing failure is presented. This step also describes the propagation of the rate and duration and formalizes the propagation criteria.

1.4 Dissertation Organization

This dissertation is organized as follows. The works that have been done and are being done by other researchers or groups on fault propagation are described in Chapter 2. Chapter 3 introduces the definition of an input failure, an artifact and the construction of the artifact. In Chapter 4, we describe the propagation phenomenon, the cancellation mechanism and the propagation criteria for the incorrect input value

failure. We also present the Image Reconstruction Method, a method developed to quantify the propagation probability of the value failure, including the procedure on which the model is based and an analysis of its efficiency. Chapter 5 explains how other value related failure modes (such as type, range and amount) are converted to the value failure or detected before the transformation. The propagation behavior of the time related failure modes (such as timing, rate, and duration) is explained in Chapter 6. In Chapter 7, we apply the method to an example. In the final Chapter, the limitations of this research and possible future research directions are stated.

1.5 Contributions

The major contributions of this dissertation are as follows:

1. Provides a procedure to construct the artifact that is used as the reference point for identification of the input failure.
2. Presents a procedure to generate the operational profile of the input failure for the software component in the system.
3. Reveals the major contributors to the masking of the fault. Presents a method to estimate the propagation probability for the incorrect input value failure mode. This method is more efficient than statistical testing and can be applied to any location in the software instead of only the input.
4. Presents a computational model for the propagation of timing failures defined in collaboration with Dr. Manuel Rodriguez that

can explain different behaviors for the “too late” and “too early” failure.

5. Formalizes the propagation of duration failure as well as rate failure in different modes and with different buffer behaviors. These modes and buffer behaviors are commonly used in real-time systems.

6. Describes how the other input failure modes such as amount failure, type failure, and range failure are transformed to the value failure.

2 Review of Fault Propagation Analysis

2.1 *Importance of Fault Propagation*

Faults remaining in software may or may not cause a system failure. Indeed, the location that contains the fault may never be executed, or alternatively, the fault is masked somewhere in the software and consequently will not propagate to the output. Fault propagation is the set of mechanisms that will decide whether a defect in the software will cause a software failure. Fault propagation analysis is important in software engineering. It is used to predict where the fault may be located, and is useful for sensitivity and testability analysis. The sensitivity of a location is estimated from the execution, infection and propagation analysis at that location. Fault propagation probability is an important characteristic of the software/system since it directly relates to the software/system reliability. The probability that the software/system fails due to a fault can be expressed as the product of the probability that the fault is executed and its propagation probability. Summing up the contributions from different types of faults, one can obtain the unreliability of the software/system. Then, the reliability of a software component can be expressed as:

$$R_{SW} = 1 - \sum_{i=1}^{N} p(FM_i)\, pp(FM_i) \qquad \textbf{(Eq 2-1)}$$

Where

R_{SW} is the reliability of the software component under study,

$p(FM_i)$ is the probability that the ith failure mode occurs,

$pp(FM_i)$ is the probability that the ith failure mode propagates to the output,

N is the number of failure modes which can occur in the software component.

2.2 *Fault Propagation Analysis*

The process of fault propagation is complex and so far has not been clearly described. Most of the research to date has focused on the study of the propagation of the data state error. A data state error occurs when a value for one variable is different from what it should be. Michael and Jones [11] found that data state errors propagate homogeneously, "*for a given input, it appears that either all data state errors injected at a given location tend to propagate to the output, or else none of them do*." Thompson et al. [12-14] introduced the RELAY model to identify the propagation flow and the corresponding propagation conditions. The model provides insight into testing and fault detection, but it is extremely complex and may not be practical for the full analysis of real programs.

A more general model called PIE was proposed by J. M. Voas [15]. In this model, the three conditions under which a fault will cause a failure are defined as 1) the fault must be executed (E), 2) execution of the fault must result in a data state error (I), and 3) the data state error must affect the remaining computation to cause an output failure (P). The PIE idea was applied in sensitivity analysis and testability analysis by J. Voas, L. J. Morell, K. W. Miller et al. [15-21]. Hiller et al. [22-24] introduced the concept of error permeability as the probability that an error in an input signal may permeate to one of the output signals. The authors used fault injection techniques to measure the error permeability in their experiments. Both Voas' and Hiller's groups studied fault propagation in the implemented code, i.e.,

they studied the propagation behavior at the code level. Nassar et al. [25, 26] studied propagation at the architecture level. They defined and estimated error propagation probabilities throughout an architecture using a generic matrix of entropies. Kao, Tang and Iyer [27] studied the hardware fault propagation as well software fault propagation using fault injection in the Unix system. Their experiment showed that the pointer faults tend to crash the system immediately if they are activated.

Some researchers also contributed to the fault propagation although their works do not address directly to the topic and therefore they did not provide clear description. Guan and Graham [28] proposed an algorithm for locating failure sources by using knowledge of device structure and fault propagation paths, and through the use of sequential testing of system sub-devices.

The following sections describe the major works which have been done in the field of fault propagation.

2.2.1 Homogeneous Propagation

In 1997, C.C. Michael and R.C. Jones conducted an experiment to study an important aspect of software defect behavior: the propagation of data state errors. They analyzed the behavior of three programs: dnet, b737, and nethack. Dnet is a small program which contains 439 lines of code (LOC). B737 is part of an automatic pilot system which contains 2,045 LOCs. Nethack is a terminal-based game containing about 75,000 LOCs. Their results show that data-state errors appear to have a property that is quite useful when simulating faulty code: for a given input, it appears that either all data state errors injected at a given location tend to propagate to

the output, or else none of them do. The results revealed that the error propagation may be homogeneous.

The error propagation is homogeneous when different data-state errors in the same variable have the same effect on the program's behavior for a given input. Generalizing this concept, one can say that homogeneous propagation occurs for any data-state error if (a) all of the errors propagate to the output, or (b) none of the errors propagate to the output.

These results are interesting because of what they indicate about the behavior of data-state errors in software. They suggest that data state errors behave in an orderly way, and that the behavior of software may not be as unpredictable as it could theoretically be. Additionally, if all faults behave the same for a given input and a given location, then one can use simulation to get a good picture of how faults behave, regardless of whether the faults one has simulated are representative of real faults.

2.2.2 RELAY Model

The RELAY model was introduced by Thompson, Richardson, et al. [12-14]. The RELAY model analyzes how a faulty code leads to a failure during the execution of some test data. In the model, the original faulty code is introduced at a location; the faulty code may trigger a potential failure and the failure may be transferred to the output.

The transfer happens along the information flow chain. The chain could be represented as a sequence of tuples $(u[x], d[x], n[x])$, where x is the number of tuples in the chain; n[x] represents the location; u[x] represents the variable used in n[x];

and d[x] represents the variable defined in location n[x]. If a node appears more than one time in the chain due to loops, it would be distinguished by a subscript. For example, one chain from node 3 to node 11 in Figure 2-1 could be represented as:

(*,E,3),(E,F,4),(E,BP,5),(BP,B,6),(B,D,10),(D,out,11).

Obviously, in one path, there could exist more than one information flow chain. For example, the chain (*,E,3), (E,F,4), (E,BP,5), (BP,C,7), (C,D,10), (D,out,11) shares the same path with the chain above.

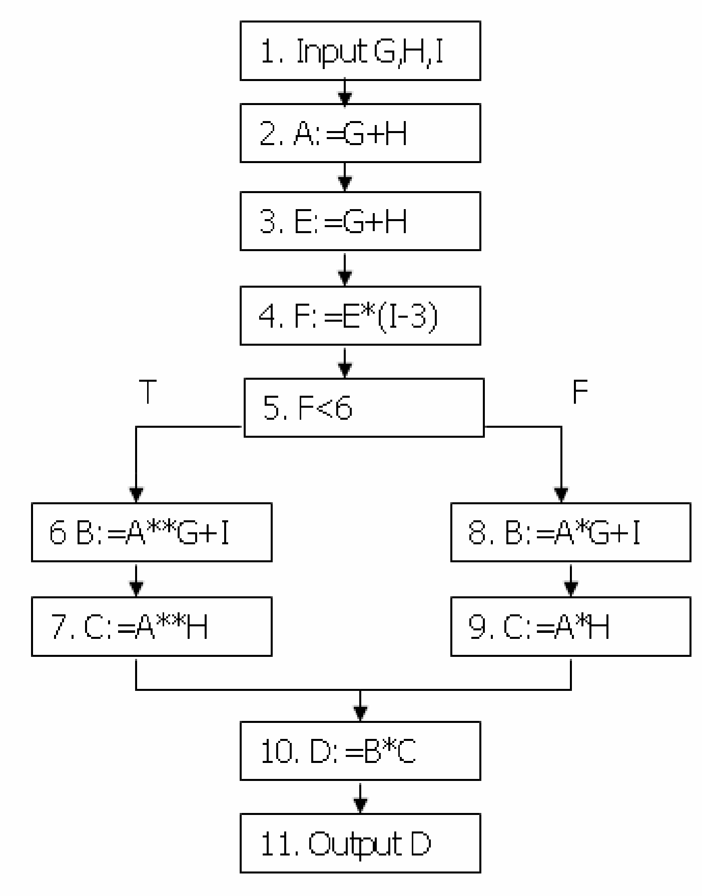

Figure 2-1 Example Module

With the information flow chain, one can find out the transfer condition. For a single chain (see Figure 2-2), the transfer condition is easy. "*The condition to guarantee the transfer along a single chain from some faulty node to a particular failure node is the conjunction of the conditions to transfer the potential failure within each node in the chain along with the condition to execute the chain.*"[12]. The necessary and sufficient condition to guarantee transfer within a single node is call computational transfer condition (ctc). The information must follow along the chain according to the definition. If any node in the chain 'breaks' (computational transfer condition is not satisfied), then the potential failures will be absorbed in the node, which is an example of coincidental correctness.

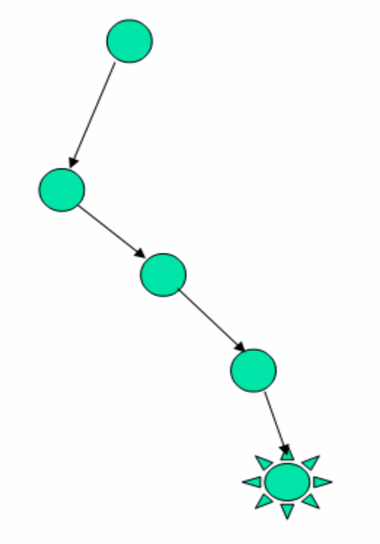

Figure 2-2 Single Chain

For Multi-chain (see Figure 2-3), the condition is more complicated. The condition for single chain is not necessary and not sufficient. The potential failures from different chains may be disappear in the nodes where they meet. In other words, it makes the problem complicated if more than one failure exists in one node. Simply,

in one code, when we calculate C =A+B, if we get A wrong with a value of 5 instead of the correct value of 2, and get 2 instead of 5 for B, then we will still get a correct result for C (7).

According to Thompson, transfer set condition could be derived in 4 steps:

1). Construct computational transfer conditions (ctc).

2). Construct transfer route condition (trc).

3). Construct transfer set path condition (pc).

4). Construct transfer set condition (tsc).

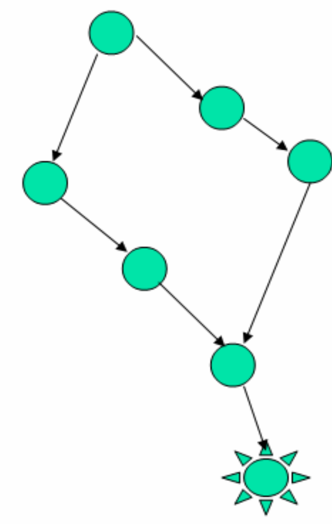

Figure 2-3 Multi-Chain

The RELAY model is useful for deciding how to select testing data. For the small program, especially those with single information flow chain; it is easy to get the transfer condition with the RELAY model. For those with multi-chain, it will be more difficult to get the transfer condition. In the large program, it will be extremely complicated and impracticable. The RELAY model is therefore not applicable for the quantity analysis.

2.2.3 PIE Model

The PIE Model is proposed by Jeffrey M. Voas in 1992. It is a dynamic technique for statistically estimating three program characteristics that affect a program's computational behavior. PIE analysis uses program instrumentation, syntax mutation, and changed values injected into data states to predict a location's ability to cause program failure if the location were to contain a fault.

PIE stands for propagation, inject and execution. The execution probability is the probability that a location is executed. An injection probability is the probability that a change to the source program causes a change in the resulting internal computational state. The propagation probability is the probability that a forced change in an internal computational state propagates and causes a change in the program's output.

2.2.3.1 Execution Probability

Execution analysis is a method that is based on program structure. As such, execution analysis is related to structural testing methods. The execution probability is decided by the proportion of inputs that cause location *l* to be executed.

2.2.3.2 Infection Probability

Infection analysis reveals statistical information about the effect that mutants have on data state. It estimates an infection probability for each mutation. It is defined as the probability that a change to the source program causes a change in the resulting internal computational state. From the definition, the infection probability is related with location, input domain and the injected fault. Different locations have different

infection probability. The infection probability can be obtained by mutation testing [29-31].

2.2.3.3 Propagation Probability

Propagation probability is defined as the probability that a forced change in the internal computational state propagates and causes a change in the program's output. To obtain the propagation probability, Voas used a perturbation function to change the data state and then counted the proportion of those perturbations that cause a change in the output.

2.2.4 Error Permeability

Similar with propagation probability, Hiller, Jhumka and Suri introduced the concept of *Error Permeability* [22, 23] and applied this concept to the Propagation Analysis Environment (PROPANE) [32].

In their work, error permeability is defined as the conditional probability of an error occurring on the output, given that there is an error on the input. Then, for input i and output k of a module M, the error permeability is defined as:

$$0 \le P_{i,k}^{M} = \Pr\{\text{err in o/p} \mid \text{err in i/p}\} \le 1$$

From the error permeability, a set of related measures can be calculated. The measures calculated from the error permeability allow for an assessment of the vulnerability of software and its modules. After getting the error permeability for each module in the system, one could use Output Error Tracing or Input Error Tracing to analysis the propagation. Furthermore, the measures and analysis results are shown to be useful input to the process of selecting locations in the software

would be suitable for error detection mechanisms and error recovery mechanisms. These methods can also pinpoint critical signals and paths in a system.

2.2.5 Fault Propagation in Software Architecture

Nassar, Ammar, etc, studied the fault propagation in the software architecture level. In their work, the *Error Propagation Probability* form component A to component B, EP(A,B), is defined as:

$$EP(A,B) = P([B](x) \neq [B](x') | x \neq x')$$

Where [B] denotes the function of component B, and x is an element of the connector X from A to B. [B] is interpreted to capture all the effects of executing component B, including the effect on the state of B as well as the effect on any outputs produced by B.

The definition of the error propagation given above uses the concept of conditional probability, i.e., one calculates the probability that an error propagates from A to B under the condition that A actually transmits a message to B. To bridge the gap between the conditional error propagation and the unconditional error propagation, denoted by E(A, B), the authors introduce the *transmission probability matrix* T. Its entry, $T(A,B)$, reflects the probability with which the connector $(A \rightarrow B)$ gets activated during an execution. Then, the unconditional error propagation can be expressed as $E(A,B) = EP(A,B) \times T(A,B)$. To estimate the error propagation probability, the authors used the formula:

$$EP(A \rightarrow B) = \frac{1 - OR(V_{A \rightarrow B}, S_B)}{1 - IR(V_{A \rightarrow B})}$$

Where the term $OR(V_{A \to B}, S_B)$ measures the average amount of redundancy in messages that B receives from A as seen by component B; i.e., based on the state transitions of component B, and hence we call this metric the observed redundancy (OR). The term $IR(V_{A \to B})$ measures the intrinsic amount of redundancy in the messages that B receives from A; i.e., the amount of redundancy in messages from A to B as seen by component A, and hence we call this metric the intrinsic redundancy (IR). The authors had confronted the results of the analytical study against the results of the empirical study to assess the validity of the analytical formula.

3 Input Failure and Artifact

3.1 Definitions

A "system" is an interdependent group of people, objects and procedures constituted to achieve defined objectives of some operational role by performing specified functions.

A complete system includes all of the associated equipment, facilities, material, computer programs, firmware, technical documentation, services, and personnel required for operations and support to the degree necessary for self-sufficient use in its intended environment. In a system, each component is connected with another component through an input-output channel. Figure 3-1 describes a 6-component system with its connections.

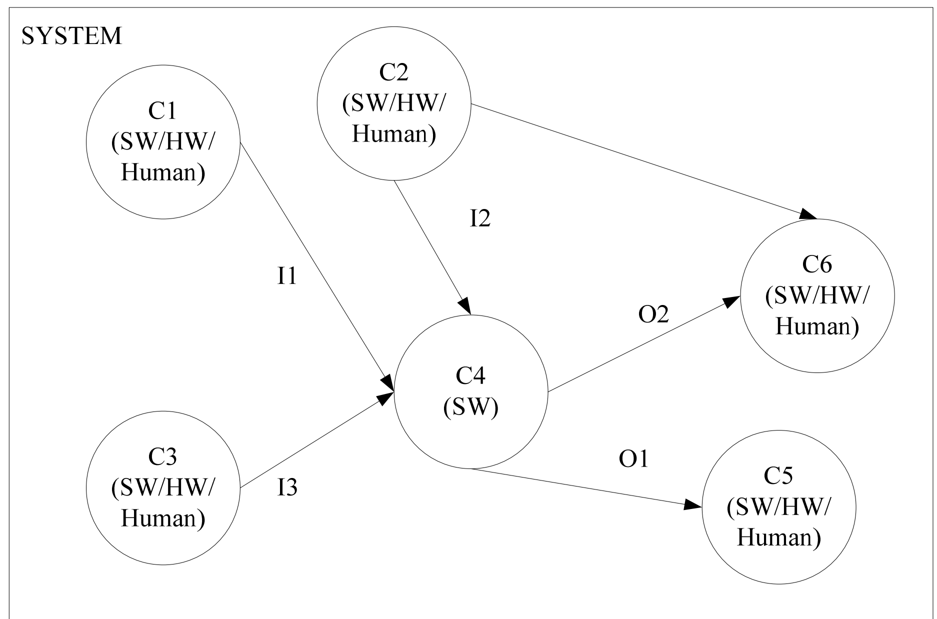

Figure 3-1 Input to a Software Component in a System

A System Requirements Specification (SyRS) is a structured collection of information that embodies the requirements of the system. A System Requirements

Specification (SyRS) has traditionally been viewed as a document that communicates the requirements of the customer to the technical community who will specify and build the system. The collection of requirements that constitutes the specification and its representation acts as the bridge between the two groups and must be understandable by both the customer and the technical community.

An "input" at time *t*, denoted as I^t, to a software component in a system is the information (*D*) received by the software component from other components in the same system, i.e., $I^t = I_1^t \cup I_2^t \cup ... \cup I_{Ni}^t$, where I_i is the input from the *i*th component, N_i is the number of components in the system that can provide information to the software component. The information can be a set of values, a stimulus, or an event.

An "output" at time *t*, denoted as O^t, from a software component in a system is the information provided by the software component to other components in the same system, i.e., $O^t = O_1^t \cup O_2^t \cup ... \cup O_{N_o}^t$, where N_o is the number of components in the system to which the software component provides information.

An "input failure" is identified right before the input goes through the software. As in the Figure 3-2, the input failure is identified at point A.

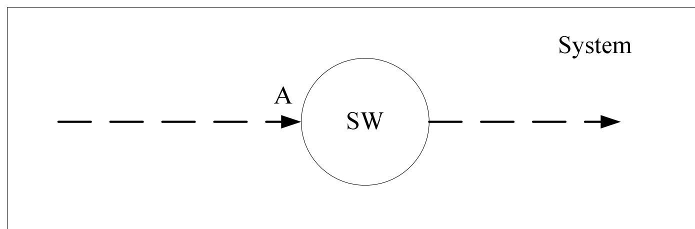

Figure 3-2 Location where input failure occurs

A reference point is required to identify the input failure. What reference point should one use? It is natural to assume that the reference point may be the system requirements specification.

For a safety-critical system, some states are defined as safe while others are defined as unsafe. In the following, the inputs that lead the system to safe states are defined as safe inputs, the remaining inputs are defined as unsafe inputs. To reach the safety goal, the corresponding reactions and operations on the inputs are described in the system requirements specification (SyRS).

Now, consider a perfect SyRS. By perfect, it is meant that:

1. (Completeness) All possible inputs are predicted.
2. (Correctness) The corresponding operations for each input are described correctly. This means that for any given input, an implementation of the system that perfectly replicates SyRS will lead the system to safe states.

Then the input domain can be represented as Figure 3-3, where A is the set of all inputs, R is the set of inputs defined in SyRS which are expected to lead the system to a safe state, S is the set of inputs which actually lead the system to safe states. The representation shows that under an assumption of perfect SyRS, $A=R=S$.

However, a perfect SyRS does not exist. SyRS may be incomplete or its statements incorrect. For an imperfect SyRS, if the SyRS is complete but partially correct, the input domain can be modified as Figure 3-4.

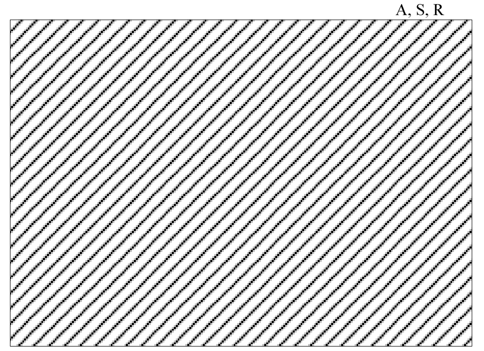

Figure 3-3 Input Domain with Perfect SyRS

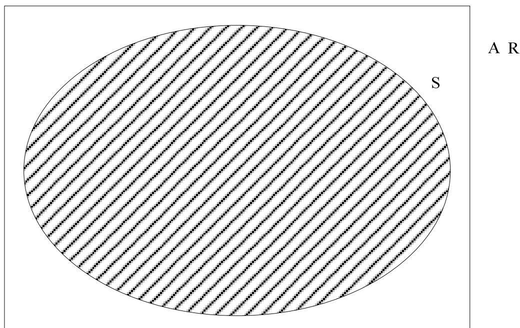

Figure 3-4 Input Domain with Complete and Incorrect SyRS

If correctness is guaranteed, then all the inputs in *R* lead the system to safe states. Those inputs which are not in *R* may or may not lead the system to safe states. Hence, the Venn diagram can be modified as Figure 3-5.

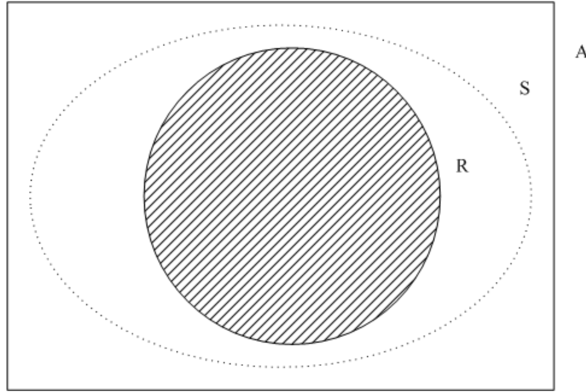

Figure 3-5 Input Domain with Incomplete and Correct SyRS

It must be noted that, it is assumed here that no other failures occur. Otherwise, the inputs defined in SyRS may cause system failure and make the investigation more complex.

Normally, the SyRS is incomplete and not all statements are correct. Some inputs defined in R cause system failure. Then, the Venn diagram is modified as Figure 3-6.

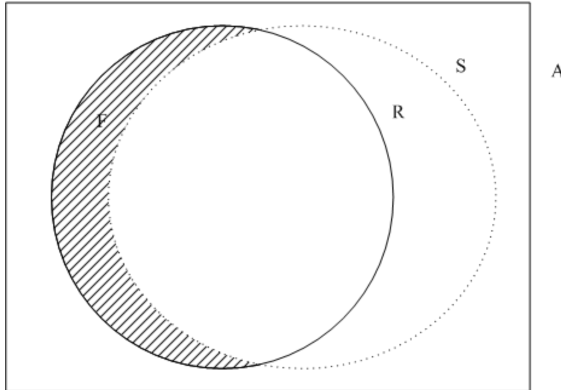

Figure 3-6 Typical Input Domain

The shadowed partition (Area *F*) is the set of inputs defined in SyRS which are expected to lead the system to safe states but actually cause system failure. The partition *F* represents the failures caused by an incorrect SyRS (Requirements Failures).

The existence of requirement failures in SyRS prevents us from identifying input failures solely on the basis of SyRS. Additional information is required to help construct a faultless reference point based on artifacts. By faultless, it means that we trust the artifact completely. Once such specification is obtained, input failures can be defined clearly.

An "expected input" at time t, denoted as $\mathscr{I}_e^t$, is a set of inputs which are defined in the faultless reference point artifact at time t, i.e., $\mathscr{I}_e^t = \{D_1, D_2, ..., D_M\}$, where M is the number of possible expected inputs ($M \geq 1$). Whether or not the system will reach a safe state depends on the downstream components. If no further

failures occur in the downstream components, then the system will reach a safe state. Otherwise, there is potential for an unsafe condition.

An "Input Failure" (or Input Fault) occurs when the actual input is different from the expected input, i.e., $\mathscr{I}^t \neq \mathscr{I}_e^t$.

3.2 *Examples of expected input and input failure*

Consider the case of PACS and a fire scenario. Assume the system was designed in such a way that the guard may override the gate in case of extraordinary conditions if he/she so desires. The expected input for the system is, thus, the "Override by Guard" command or equivalent input. Another issue is that the override command should be received in time (with the respect to the dynamics of fire) to allow people in the building enough time to safely exit.

Assume that if the guard override command arrives before a time limit t_{LIMIT}, the system is safe, i.e., all employees and visitors are safe. Also, assume that the software is such that it allows inputs only during time windows of T ($T > t_{LIMIT}$) seconds. The software and system time window is shown in Figure 3-7.

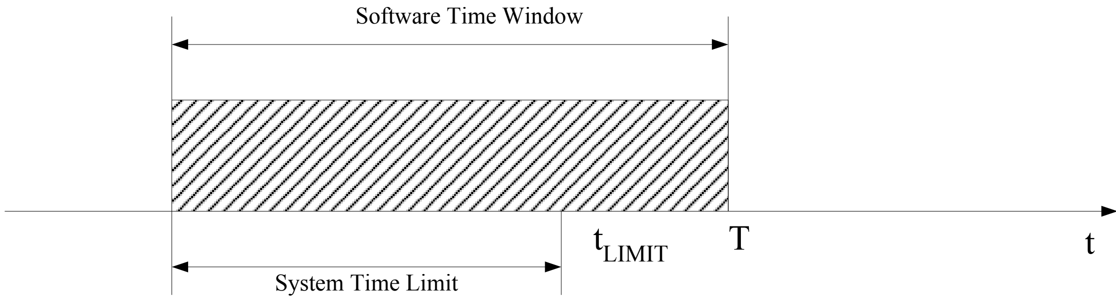

Figure 3-7 Software and System Time Windows

In this case, the expected input is: “Guard Override,” $0 <= t < t_{LIMIT}$ and an input failure occurs if the actual input to the software arrives at a time superior to t_{LIMIT}. Although in the software SRS, an input within the interval $[t_{LIMIT}, T)$ is allowed by the software component, such input will cause a system failure. Hence, the input: “Guard Override”, $t_{LIMIT} <= t < T$ is not an expected input.

3.3 *Construction of the Artifact*

The artifact is a combination of software requirements specification (SRS), system requirements specification (SyRS), implementation documents, testing reports, historical data and expert opinions.

- System Requirements Specification (SyRS)

  A System Requirements Specification (SyRS) has traditionally been viewed as a document that communicates the requirements of the customer to the technical community who will specify and build the system. The collection of requirements that constitutes the specification and its representation acts as the bridge between the two groups and must be understandable by both the customer and the technical community.

  SyRS describes the role of the software component in the system and its interconnection with other components in the system. SyRS will help to determine if the Software Requirement Specification (SRS) describes the software behavior correctly. It is a major component of the artifact.

- Software Requirement Specification (SRS)

An SRS is basically an organization's understanding (in writing) of a customer or potential client's software requirements and dependencies at a particular point in time (usually) prior to any actual design or development. It is a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

Normally, an SRS will address nine topics: interface, functional capabilities, performance levels, data structures/elements, safety, reliability, security/privacy, quality, and constraints and limitation. The SRS describes the procedure by which a software component responds to inputs provided by the system. It is another major component of the artifact.

- Hazard Analysis-Related Documents

  Hazard Analysis helps identify hazard scenarios that could adversely affect people, property, or the environment.

- Implementation Documents

  The implementation documents include the high level design documents and detailed design documents. These documents help to determine if the software component is designed correctly according to the SRS.

- Testing Reports

  Test reports contain the required behaviors of the software for every input in the test cases as well as the required output. They thus contain valuable system and software requirement information which may not appear directly in either SyRS or SRS.

- Expert Opinions

  Not all abnormal situations have been predicted in the foregoing documents. Expert opinions may provide helpful suggestions as to situations not covered in these documents based on accumulated knowledge.

- Historical data

  Historical data is the information collected from failures reports describing the operation of a previous release of the system or of similar components in other safety critical systems.

- Results from simulation

  Simulations are performed in the design stage or even in the execution stage if needed. They provide us an image of system operation.

The major issue related to creation of the artifact is the systematic elimination of incorrectness and inconsistencies between documents. Also, the incompleteness of the SyRS and SRS should be remedied using the supplemental information discussed above. The steps below can be applied to refine the documents at hand:

1. Check the correctness of SyRS. The criterion used to determine correctness is whether or not the safety goal is met. In SyRS, the functionalities of all the components and their interactions are described. Check if the functionalities of the components meet the safety goal. Fix the procedures which are determined to be incorrect.
2. Check the SRS according to the SyRS. The purpose of this step is to 1) check the completeness of functionalities of the software components according to SyRS, 2) check the consistency between the output of the software and other

components connected with the software, and check the correctness of the descriptions about the software procedures.

3. Check the implementation documents according to the SRS. The step is to 1) check the correctness of the implementation documents according to the SRS, and 2) record the redundant or additional functionalities. The redundant or additional functionalities are introduced by the designers or developers for some purpose. They do not influence the software component behavior according to the SRS, but they may cause software component failure in some special scenario that is unknown at this point.
4. Investigate the testing reports. Check if the results are correct according to the implementation documents.
5. Check the completeness of SyRS. Investigate the historical data; if there are some scenarios that may cause system failure and are not addressed in SyRS, append those scenarios and the corresponding handling procedures. Determining the corresponding handling procedures may require referring to expert opinions and lessons learned.

Steps 1-5 are typically performed as part of the software quality assurance activities that accompany the development of a software code.

3.4 *Construction of the Operational Profile*

An *operational profile* is a description of distribution of inputs that may occur during software operation.

It is very difficult to generate the operational profile for the software components, especially for the sub-software components. To estimate the propagation probability, two $\mathcal{OP}$s are required: $\mathcal{OP}_f$ and $\mathcal{OP}_e$, represents the operational profile for the input failure and expected input respectively.

Generally, the operational profile for the input failures is defined as $\{\mathcal{I}, P(\mathcal{I})\}$, where $\mathcal{I}$ is the set of input failure and $P(\mathcal{I})$ is the set of probabilities of $\mathcal{I}$. Hence, to construct an operational profile for an input, two steps should be performed:

1. Determine the possible value for the input.
2. Determine the probability for each value.

In the following sections, we discusses the operational profile generation for expected input and input failure in turn.

3.4.1 Generation of the expected input I_e

To determine the expected inputs space for the software component, one has to rely on the artifact. The process of retrieving expected inputs from the artifact is given below. The behavior of the upstream components defines the operational conditions, PIi for the ith software component. Given these conditions, the artifact returns the expected inputs. Consider the artifact as one database, the process to obtain expected inputs can be expressed as:

$$\begin{aligned}\mathcal{I}_{e,i} \leftarrow & \text{SELECT } \textit{Expected Inputs} \\ & \text{FROM } \textit{Artifact} \\ & \text{WHERE } (\textit{Previous Info} = PI_i)\end{aligned}$$

Where,

$\mathcal{I}_{e,i}$ is the set of expected inputs for the ith software component,

PI_i represents the previous operation information before the ith software component is executed.

For instance, in PACS, one of the scenarios found in the ESD for the fire initiator is *"the fire protection system failed; emergency exit failed; guard was not there; and a user's (Yuan) card was swiped (as shown in Figure 3-8)." We also know that the fire started 20 minutes ago.* Now, what is the expected input for the software component "Enter PIN"?

The Artifact is composed of safety analyses results and the SRS.

The Artifact (i.e., here, the safety analysis results) shows that "If the fire started 20 minutes ago, only 5 minutes are left for Yuan to safely exit the building."

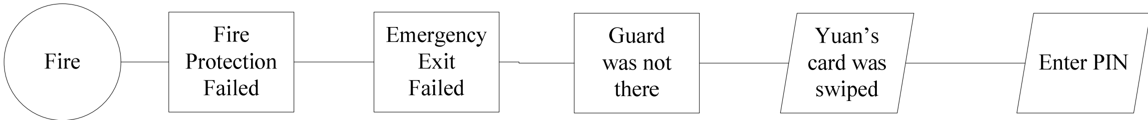

Figure 3-8 Example Scenario

To further define the expected input, we turn to the SRS (the second part of the Artifact) and find the following information:

If all four digits from the PIN have been entered, the software will compare it with the PIN number which was retrieved from the security file during the reading of the ID card. On successful comparison, the software will open the gate (described later); otherwise, it will post the message "INVALID PIN" to the user's display.

The previous operation information before the software component "Enter PIN" is that "Yuan's card is swiped." We check the security file stored in the system. Assume the file for valid users is written as Table 3-1. The corresponding PIN can

then be retrieved. This PIN is 2222. Combining this information with the information retrieved from the safety analyses results, one can obtain the expected input (PIN = 2222, before 5 minutes).

Table 3-1 Example Security File

Card		
SSN	**Name**	**PIN**
111111111	Yuan	2222
222222222	Manuel	3333
………	………	………

The probabilities for the expected inputs will be determined along with the generation of $\mathcal{OP}_f$ discussed in the following sections.

3.4.2 Generation of an Operational Profile for Input Failures

The input comes from the upstream components: hardware components, software components, or human components. $\mathcal{OP}$ can be therefore considered as the combination of output distributions from hardware, software and human, denoted as $\mathcal{OP}_{hw} = \{(I_{hw}, P(I_{hw}))\}$, $\mathcal{OP}_{sw} = \{(I_{sw}, P(I_{sw}))\}$, and $\mathcal{OP}_{human} = \{(I_{human}, P(I_{human}))\}$ respectively.

3.4.2.1 Generating $\mathcal{OP}_{hw}$

Hardware failures occur due to incorrect design, manufacture or wear down. To generate the operational profile contributed by the upstream hardware components, the following steps may be useful:

Step 1. Identify the upstream hardware components which may provide inputs to the software component.

Step 2. Find the failure mechanisms for each hardware component from the hardware manufacturers or from the market reports.

Step 3. Model the behavior of the hardware components for each failure mechanism and the normal behavior without any hardware failures.

Step 4. Determine the input failure space for the software component and quantify the probabilities with the model developed in step 3 and the operational profile of the hardware component.

The detailed description of each step is discussed in turn in the following paragraphs.

Step 1. In this step, only hardware components, which provide inputs to the software component directly, as defined in the system abstraction, are considered.

Step 2. The failure mechanisms for the hardware component may not be noted in the hardware user manual. To find out all the failure mechanisms, one should contact the manufacturer or collect information from the market reports like products review. These types of documents contain description about the possible failure mechanisms and their failure rates.

Step 3. This is the most difficult step. The normal behavior model can be developed according to the functionality of the hardware component. Among the hardware failure modes, some may cause system failure directly. Modeling the behaviors for different failure modes may require help from hardware designers.

Step 4. Once the models developed in step 3 are obtained, one can feed the operational profiles of input to the hardware components into the model. Record their outputs and associated occurrence and generate the operational profile for each mode. Then, the input space from hardware, donated as $\Im_{hw}$, and the probability distribution

can be obtained by combining all the sub operational profiles of all the hardware failure modes. The whole process is described in Figure 3-9.

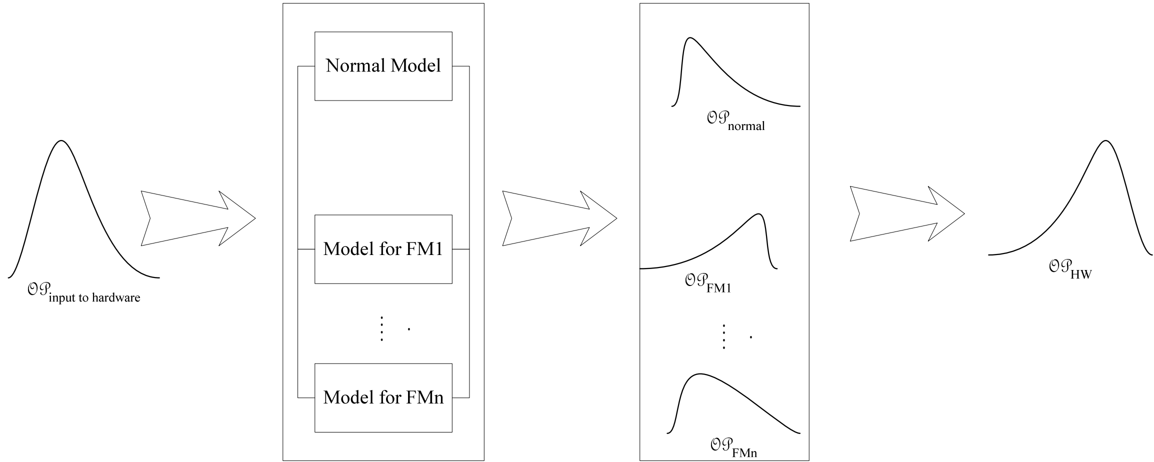

Figure 3-9 Generating $\mathcal{OP}_{hw}$

3.4.2.2 Generating $\mathcal{OP}_{sw}$

Generating $\mathcal{OP}_{sw}$ is similar to generating $\mathcal{OP}_{hw}$. To represent software behavior, two models should be considered. The first model corresponds to the behavior of the software in the absence of support failures. Such a model representates the software implementation or the actual code. The second accounts for support failures. During software execution, support failures may arise that will cause changes in software behavior. The process of generating $\mathcal{OP}_{sw}$ is shown in Figure 3-10.

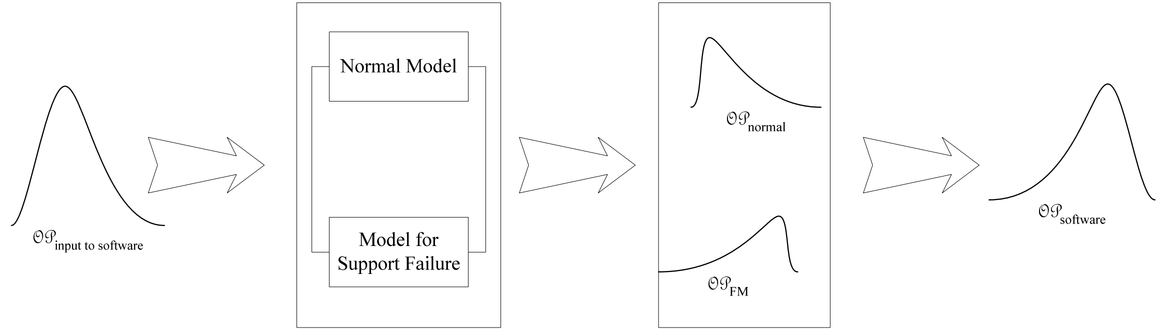

Figure 3-10 Generating $\mathcal{OP}_{sw}$

3.4.2.3 Generating $\mathcal{OP}_{\text{human}}$

Actually, human failures influence the software component through hardware components. Methods to model human behavior include cognitive models, human reliability models. Humans provide input to software components through hardware components (like keyboard and mouse). Hence, the failures of such hardware components should be considered when building the human models. This means that both the normal and abnormal hardware behavior should be modeled into the normal and abnormal human behavior model. In addition, one can use the results of expert opinion or experiments to build the profile.

3.4.2.4 Generating $\mathcal{OP}_e$ and $\mathcal{OP}_f$ from $\mathcal{OP}_{hw}$, $\mathcal{OP}_{sw}$, and $\mathcal{OP}_{\text{human}}$

The input space to a software component is the Cartesian product of input spaces from the upstream hardware, software, and human components, i.e.,

$$\mathcal{I} = \mathcal{I}_{hw} \times \mathcal{I}_{sw} \times \mathcal{I}_{human}$$

Where

$\mathcal{I}_{hw}$, $\mathcal{I}_{sw}$, and $\mathcal{I}_{\text{human}}$ are the input spaces from the hardware, software, and human components respectively.

Similarly, the operational profile is given as:

$$\mathcal{OP} = \{\mathcal{I}, P(\mathcal{I})\} = \{(I_{hw}, I_{hw}, I_{human}), P(I_{hw}) \bullet P(I_{sw}) \bullet P(I_{human})\}$$

This expression assumes independence between the input variables from hardware, software and human.

Given $\mathcal{OP}_{hw}$, $\mathcal{OP}_{sw}$, and $\mathcal{OP}_{human}$, one can obtain the operational profiles for input failures and expected inputs using the following two steps:

1. Subtract the expected inputs from the operational profile. The input space is divided into expected inputs space, $\mathcal{I}_e$, and input failures space $\mathcal{I}_f = \mathcal{I} \setminus \mathcal{I}_e = \overline{\mathcal{I}_e}$
2. Normalize the distributions in the expected inputs space and input failure space respectively to obtain the operational profiles for expected inputs and input failures.

The input profile for one component depends on its upstream components. When generating the operational profiles for all the components in the system, one needs to generate the operational profile for the most upstream component and then the one for the next component, etc.

3.5 Types of Input Failures

Input Failures can be divided into two groups: value-related failures and time-related failures (as shown in Figure 3-11). Value-related failures cover the characteristics such value, type, range and amount while time-related failures address

characteristics such as time, rate, and duration. For each characteristic, there are one or more failure modes.

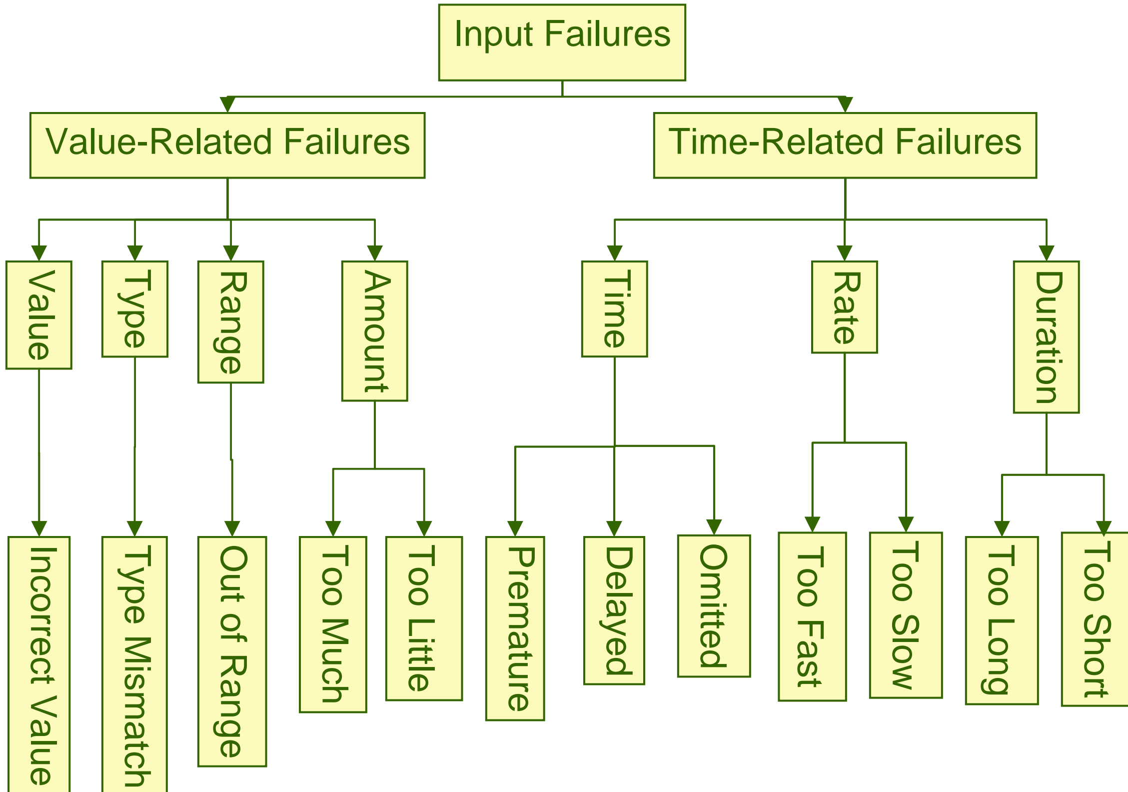

Figure 3-11 Input Failure Modes

4 Fault Propagation of Incorrect Input Value

4.1 Data State and Data State Error

The fault is transferred through the data state (or program internal state). Different researchers have provided different definitions of the data state. In [33], Daran et al. defined the program internal state as:

> *The program internal state at a point during execution is defined by the values (val) of all variables (var) and the value (x) of the program counter (PC) which indicates the next instruction to be executed, i.e.,*
>
> $PS = \{(\mathrm{var}_1, val_1), ..., (\mathrm{var}_n, val_n), (PC, x)\}$

Their definition is different from Zeil's definition for program state [34] that does not include the program counter, i.e., $PS = \{(\mathrm{var}_1, val_1), ..., (\mathrm{var}_n, val_n)\}$. Daran's definition is similar to Voas's definition for program data state [15]:

> *A program data state is a set of mappings between all variables (declared and dynamically allocated) and their values at a point during execution. In a data state we include both the program input used for this execution and the value of the program counter.*

The difference lies in the fact that, in Daran's definition, all variables of the program data state have undefined values before a program execution on an input begins. However, when studying fault propagation in a software component, these two definitions can be considered as consistent.

A data state error occurs when a variable/value pairing in a data state is different from the expected variable/value pairing that is determined by an oracle.

4.2 Information Flow Transfer

When a data state error occurs, it may affect the next data state via an information flow that includes data dependent transfers and control dependent transfers. The data dependence transfer will change some value of the variables, while control dependence transfer may change the value of the program counter that indicates the location of the next operation.

4.2.1 Data Dependent Transfer

A node q is *data dependent* on a node p if and only if there is a definition for V at node p and V is used at node q. As this example, A is defined at node 1, and B is defined at node 2 using A. Hence, B is data dependent on A. Similarly, C is data dependent on B.

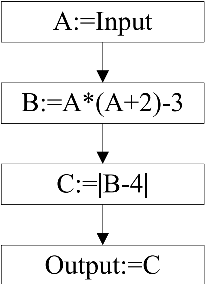

Figure 4-1 Data Dependent Transfer

4.2.2 Control Dependent Transfer

A node q is control dependent on a node p if and only if p is the conditional judgment point for q. For example, in Figure 4-2, the node “c=0” is control dependent on the node “if A>0.” And the node “c=1” is also control dependent on the same node.

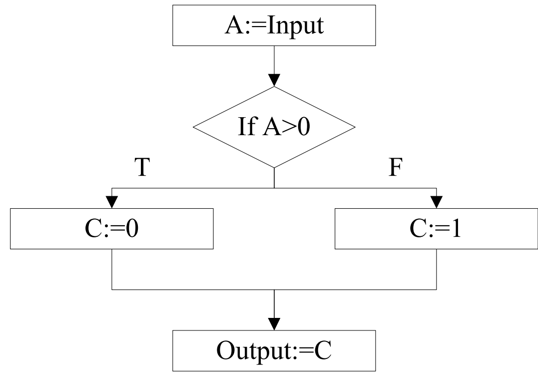

Figure 4-2 Control Dependent Transfer

4.3 Propagation Behavior

4.3.1 Error Flow Model

If no fault has executed and infected the program data state, one correct data state will transfer to another correct data state, and so forth and so on, till the correct output. If one fault is executed and infects the data state, the correct data state will be corrupted and cause a data state error. If the data state error is masked or cancelled during the remaining computation, it will transfer back to the correct data state; otherwise, it will propagate to the output and cause a failure. The model is shown in Figure 4-3.

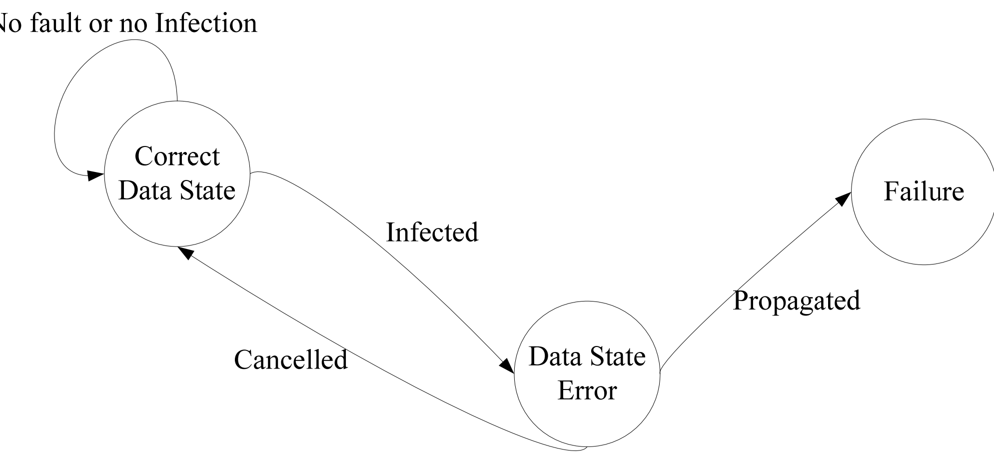

Figure 4-3 Error Flow Model

4.3.2 Cancellation Mechanisms

Cancellation is commonly referred to as coincidental correctness. For instance, variable "a" carries a faulty value -2 instead of the correct value 2. After the execution of an assignment $a = |a|$, the variable "a" contains a correct value 2. In other words, the fault has been cancelled. In software, normally, the outputs are only a subset of the total variables defined in the software. The correctness of the output variables do not indicate a correct data state. For example, in the small software below:

```
Input a;
b=|a|;
output b;
```

Variable "a" carries a faulty value -2 instead of the correct value 2. After the execution, the output variable "b" is correct. However, according to the definition of data state, the data state in the output is $\{(a,-2),(b,2),(PC,x)\}$, that is different

from the correct data state $\{(a,2),(b,2),(PC,x)\}$. *In this dissertation, if the states of all the output variables are correct, the fault is treated as cancelled.*

4.3.3 Major Contributors of Masking

When an input failure occurs, it is propagated to the output through the remaining program, *f*, iff $f(I_e) \neq f(I_f)$. The propagation is directly related with the remaining function *f*. Some functions can tolerate/mask all the faults, e.g., $f(I) = \text{constant}$, while some functions propagate all the faults, e.g. $f(I) = I$. Indeed, the propagation behavior can be classified into the following cases (as shown in Figure 4-4).

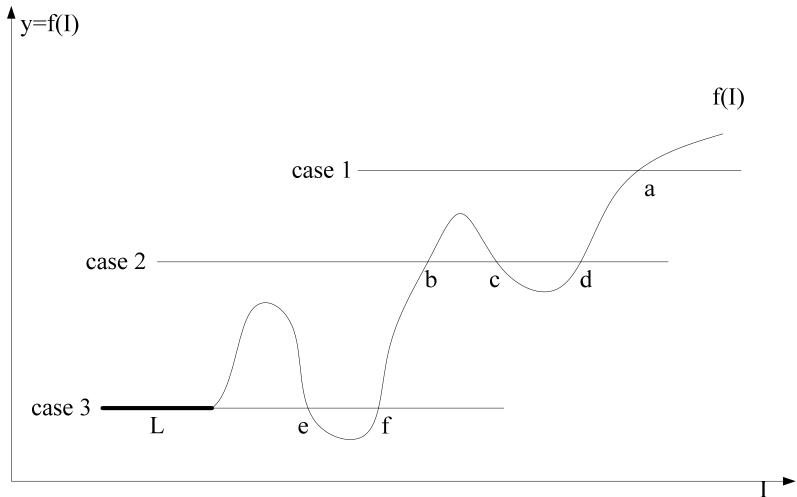

Figure 4-4 Propagation Behavior

Case 1, equation $f(I_e) = f(I_f)$ cannot hold due to the restriction $I_f \neq I_e$, i.e., there is no solution for the equation $f(I_e) = f(I_f)$.

Case 2, equation $f(I_e) = f(I_f)$ holds if I and I_e are chosen from the set {b, c, d} respectively, i.e., every (I_e, I_f) (and vice versa) pair, i.e., (b, c), (b, d) or (c, d), will hold the equation.

Case 3, equation $f(I_e) = f(I_f)$ holds if I and I_e are chosen from the points in flat part L or points e or f. The flat part means that the region is insensitive to the faulty variable x: the points in the region generate the same function value.

If the function is continuous, the probability that the input data will fall into the line L is much greater than the probability that the input data will actually be equal to the points listed in case 2 and case 3. Therefore, the flat parts of the function are the main contributors to the masking of the fault. This can be proved by the following theorem.

> **Theorem:** Given a continuous function *f*(*x*), $x \in$D, if
>
> $\forall x^* \in D,\ \exists x_i \in D,\ i = 2,3.....N, 0 \le \mathrm{N} < \infty \textit{ such that } f(x_i) = f(x^*)$,
>
> The probability that a fault is masked in the function is 0.

Proof: Assume x^* is the true value used in the function *f*(*x*), then the probability that a fault is masked in the function is equal to the probability that the false value is equal to x_i, i=1,2,…N. Consider the neighborhood $\left[x_i - \frac{\delta x_i}{2}, x_i + \frac{\delta x_i}{2}\right] = d_i$, where $0 < \delta x_i < 1$. The probability that the false data falls in the interval d_i, i=1, 2, …, N is:

$$prob = \sum_{i=1}^{N} p_i \delta x_i \le Np\delta x$$

Where:

$\delta \mathrm{x} = \max\left[\delta \mathrm{x}_1, \delta x_2 ......\delta x_N\right]$,

$p = \max[p_1, p_2......p_N]$,

p_i is the probability density in $\mathrm{x_i}$, $i = 1, 2......N$,

By taking the limit when δxi goes to 0, the probability is obtained:

$$prob = \lim_{\substack{\delta x_i \to 0 \\ i=1,2...N}} \sum_{i=1}^{N} p_i \delta x_i \le \lim_{\delta x \to 0} Np\delta x = 0$$

Although the computer can only provide a discrete representation and therefore δ*x* can never be 0, the probability above can still seen as 0, because δ*x* is extremely small and insignificant.

In fact, a computer is only capable of storing a floating-point number to a fixed number of decimal places. In every computer, there is a smallest number, η, when added to a number of order unity gives arise to a new number[1]. Hence, every floating-point operation incurs a round-off error of $O(\eta)$ which arises from the finite accuracy to which floating-point numbers are stored by the computer. Fortunately, η is small enough and can be ignored in the computation. For example, the precision for the double floating number is about 2.2×10^{-16}. Hence, the flat parts are the only regions that contribute to the non-propagation. The flat part discussed is the

[1] When the new number is taken away from the original number, the computer yields a non-zero result.

insensitive region L with regard to the faulty input variable x, i.e., $f(x_i) = f(x_j), x_i, x_j \in L$.

The concept of the flat part is extensible for the multiple input faults. If more than one input variable is incorrect, e.g., $x_1, x_2, \ldots, x_k$ are incorrect, the flat part with regard to these false input variables can be defined as region L in which $f(x_{1i}, x_{2i}, \ldots, x_{ki}) = f(x_{1j}, x_{2j}, \ldots, x_{kj}), (x_{1i}, x_{2i}, \ldots, x_{ki}), (x_{1j}, x_{2j}, \ldots, x_{kj}) \in L$. Similarly, the flat parts are the only regions that can mask multiple input faults.

4.3.4 Multi-layer Traps

A software application may contain more than one flat part distributed in different components inside. The fault which falls in one flat part may also fall in another flat part. In other words, the flat parts of one component may fall in some of the flat parts of the remaining components in the software. For example, in a two-component system, the components are modeled as:

$$component\,1 \qquad x_1 = f_1(x_0) = \begin{cases} (x_0 - 4)^2 & 0 \le x_0 \le 8 \\ 16 & otherwise \end{cases}$$

$$component\,2 \qquad x_2 = f_2(x_1) = \begin{cases} x_1 & 0 \le x_1 \le 9 \\ 9 & otherwise \end{cases}$$

In the first component, $x_0 \in [8,10]$ is the flat part and $x_1 > 9$ is the flat part in the second component. The flat part of the first component falls in the flat part of the second component because $x_1 = 16 > 9$. This phenomenon is called a multi-layer trap (see Figure 4-5). Similar to the tree data structure in computer science, each trap (flat part) has zero or more child traps (sub-traps) that come from the previous functions. A trap that has a child trap is called the child's parent trap. A trap without a parent is

called root trap. A child trap has at most one parent trap. Traps without children are called leaf traps. The existence of multi-layer traps makes it difficult to describe and quantify fault propagation.

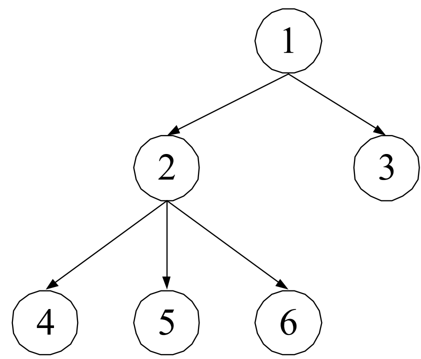

Figure 4-5 Multi-Layer Trap

4.3.5 Potential Mask Area and Potential Propagate Area

Different flat parts may have the same function value. Those flat parts with the same function value should be considered as one group. To distinguish the flat parts with different function value, we make the following definitions.

> ***Potential Mask Area (PMA):*** A Potential Mask Area is the combination of the one or more flat parts in the input domain that map to the same result in the output.
>
> ***Potential Propagation Area (PPA):*** The Potential Propagation Area is the combination of all areas in the input domain which don't contain any flat parts.

Based on the definition, an input domain may contain several PMAs and only one PPA. As shown in Figure 4-6, the input domain contains two PMAs. PMA1 consists of two flat parts while PMA2 consists of only one flat part. The area other than the two PMAs is the PPA.

Once all PMAs and PPA are identified, the propagation behavior can be determined. Whether a fault is propagated to the output is decided by the location of the function itself and the location of correct and incorrect input. Based on their location, four categories are identified.

1) Both correct input and incorrect input fall in the same PMA. Since both the incorrect input and the correct input have the same function value in the output, the fault will NOT propagate to the output.

2) Correct input and incorrect input fall in different PMAs. Based on the definition of PMA, different PMAs have different function values, and the fault will propagate to the output.

3) Both correct input and incorrect input fall in PPA. Because there is no flat part in the PPA, the propagation that they have the same output is considered as 0. Hence, the fault will be revealed in the output.

4) Correct input and incorrect input fall in one PMA and PPA respectively. The fault will be propagated as that in case 3.

Only when both correct input and incorrect input fall in the same PMA, can the fault be masked. The four categories above can be summarized as the PMA-PPA matrix (see Table 4-1). When the correct input and the incorrect input change their location with each other, the propagation behavior will not change, so the matrix above is symmetric.

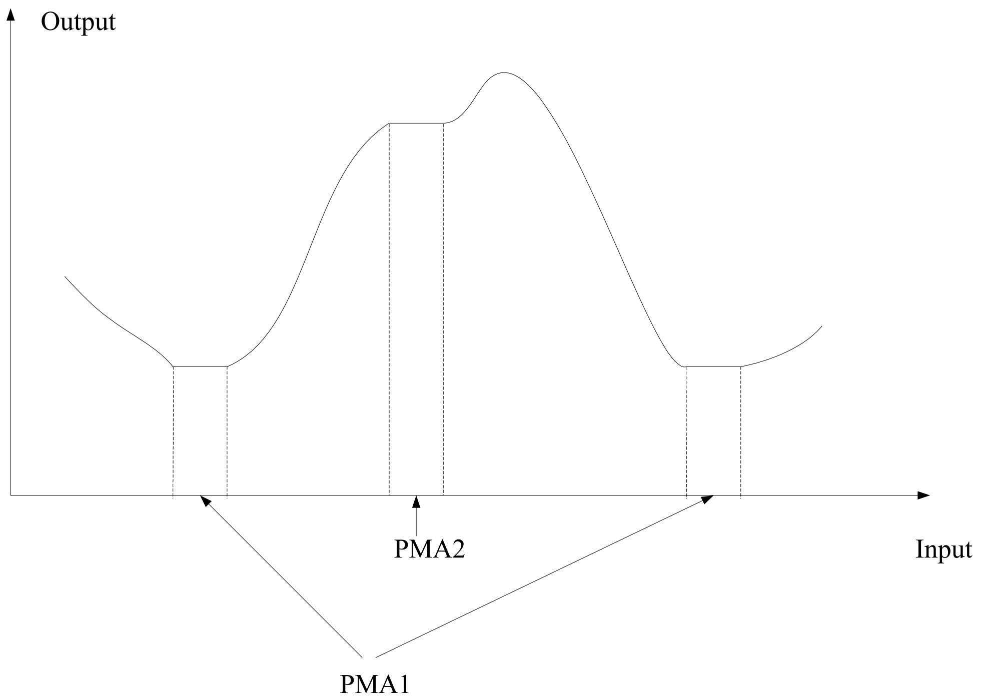

Figure 4-6 PMA&PPA

Table 4-1 PMA-PPA Matrix

	PMA_1	PMA_2	……	PMA_N	PPA
PMA_1	N	P	P	P	P
PMA_2	P	N	P	P	P
……	P	P	N	P	P
PMA_N	P	P	P	N	P
PPA	P	P	P	P	P

P: propagate N: non-propagate

Hence the propagation probability (pp) can be obtained as:

$$pp = 1 - \sum_{i=1}^{N} p_{ti} p_{wi} \quad \textbf{(Eq. 4-1)}$$

Where

p_{ti} is the probability that the correct input falls in PMA_i,

p_{wi} is the probability that the incorrect input falls in PMA_i,

N is the number of PMAs.

4.3.6 Single Fault and Multiple Faults

Normally, the input vector includes more than one variable, therefore two distinct cases should be considered: 1) Only one of the input variables is incorrect (single input fault), and 2) More than one input variable is incorrect (multiple input faults). Single faults happen more frequently than multiple faults.[2]

In the case of single input fault, the variables other than the incorrect input variable can be seen as parameters. For example, for a system with N input variables $\{x_1, x_2 \ldots x_N\}$, if x1 is the variable that carries the fault, then $\{x_2, x_3 \ldots x_N\}$ can be seen as the parameters of the system. In the output density distribution, some impulses can be seen and their corresponding probabilities are the function of $\{x_2, x_3, \ldots, x_N\}$. Therefore, the non-propagation probability (npp) of the system for the false input variable x_1 can be expressed as the integration over the parameters.

$$npp = \int npp(x_2, x_3, \ldots, x_N) dx_2 dx_3 \ldots dx_N$$
$$= \int \sum_{i=1}^{n} p_{ti}(x_2, x_3, \ldots, x_N) p_{wi}(x_2, x_3, \ldots, x_N) dx_2 dx_3 \ldots dx_N$$

In the case of multiple input faults, assuming the false input variables are x_1, x_2, …, x_k, the non-propagation probability for these multiple input variables is:

[2] If the input variables are independent of each other, the probability that N variables are incorrect at the same time is p^N (p is the probability that one variable is incorrect). Customarily, p is a small number, so $p^n << p$ (n = 2, 3 … N).

$$npp = \int npp(x_{k+1}, x_{k+2} ..., x_N) dx_{k+1} dx_{k+2} ... dx_N$$
$$= \int \sum_{i=1}^{n} p_{ti}(x_{k+1}, x_{k+2} ..., x_N) p_{wi}(x_{k+1}, x_{k+2} ..., x_N) dx_{k+1} dx_{k+2} ... dx_N$$

Where:

p_{ti} is the probability that the true value falls in the ith impulse given x_{k+1}, .., x_N,

p_{wi} is the probability that the incorrect value falls in the ith impulse given x_{k+1}, .., x_N.

4.4 Propagation Probability Estimation

4.4.1 Probability Representation and System Overview

Because all the points in one flat part share the same function value, there is a corresponding impulse in the function density distribution (whose value is equal to the probability that the point falls in the flat part). If several flat parts share the same function value, the value for the impulse is the probability that the point falls in any one of the flat parts.

However, not all the flat parts can be reflected in the output density distribution. As another two-component-system demonstrated below, in the density distribution of variable x_1, there is one impulse ($x_1 = 1$) corresponding to the flat part $0 \le x \le 1$, and in the density distribution of x_2, the impulse disappears. This phenomenon is called impulse absorption.

$$x_0 \in [0, 5]$$
$$component\ 1 \qquad x_1 = f_1(x_0) = \begin{cases} 1 & 0 \le x_0 \le 1 \\ x_0 & otherwise \end{cases}$$
$$component\ 2 \qquad x_2 = f_2(x_0, x_1) = x_0 x_1$$

To avoid handling this difficulty, the whole system is considered as one component. Looking backward from the output of the system (see Figure 4-7), some impulses will be seen in the system probability density distribution if some root traps exist in the system. These impulses are independent although their properties (single- or multi-layer) are unknown. In fact, it is not necessary to know their properties. The impulses in the output density distribution represent the root traps for the whole system and they contain complete information on non-propagation probability.

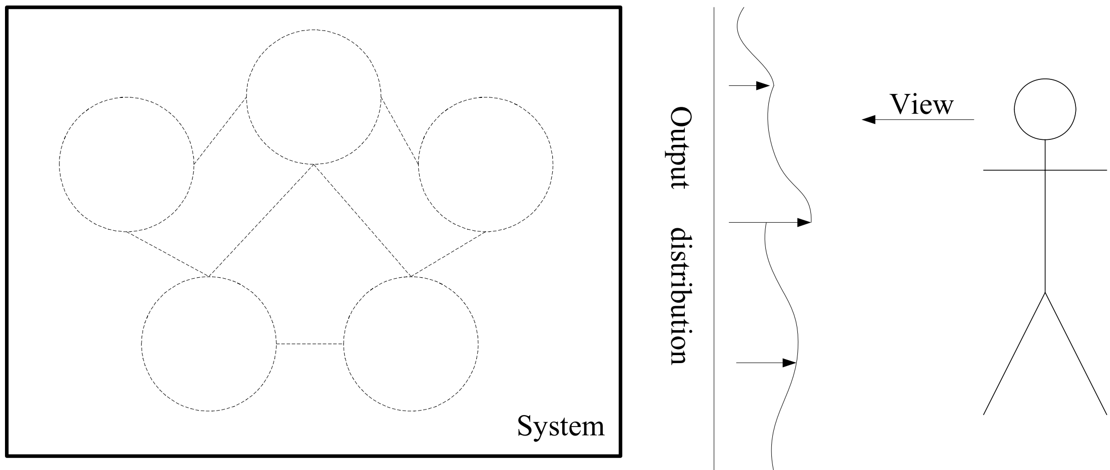

Figure 4-7 System Overview

As described in Section 4.3.3, only the flat parts contribute to the non-propagation probability. When both correct value and incorrect value fall in the same root trap, the fault will not propagate to the output. When they do not fall in the same root trap, the fault will propagate to the output. This means that the non-propagation probability in the system can be estimated through the impulses (root traps) in the output density distribution.

The impulses in the output density distribution are independent of each other, so the propagation probability can be expressed as:

$$pp = 1 - \sum_{i=1}^{N} p_{ti} p_{wi}$$ **(Eq. 4-2)**

Where

p_{ti} is the probability that the true value falls in the ith impulse,

p_{wi} is the probability that the incorrect value falls in the ith impulse,

i is the index of impulses,

N is the number of impulses.

pti is calculated using the distribution of the true value and the area in the input domain that maps to the ith impulse. pwi is calculated using the distribution of the incorrect value and the area in the input domain that maps to the ith impulse. As shown in Figure 4-8, pti and pwi are calculated using different distributions and maybe different ranges. (Eq. 4-1) and (Eq. 4-2) are identical. Only the representations of the terms are different. Actually, each impulse represents one PMA in the software.

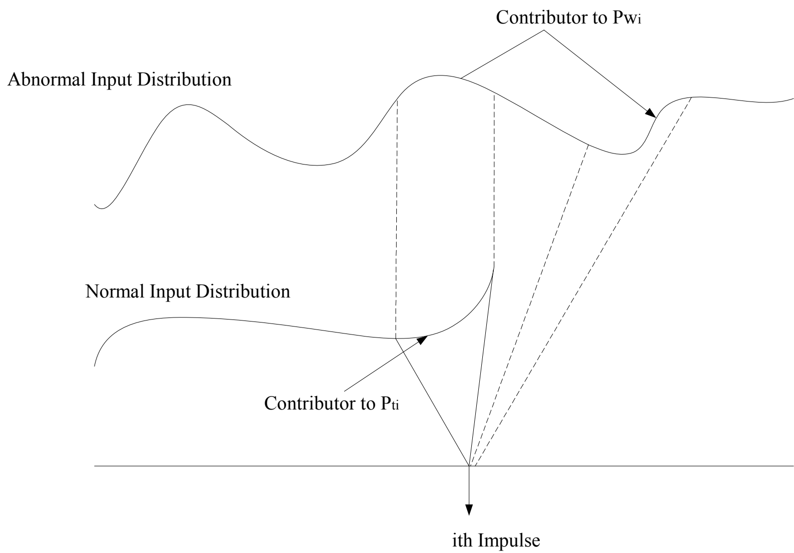

Figure 4-8 P_{ti} and P_{wi} Calculation

4.4.2 Image Reconstruction Method

As shown in the discussion above, the propagation probability can be estimated once the impulses in the output density distribution are identified. It is very difficult to get the output density distribution. However, instead of using (Eq. 4-2), one could use (Eq. 4-1) because it is easier to identify the PMAs. In this section, a 3-step method called Image Reconstruction method is suggested.

The method discussed consists of three steps: Data Generation, Image Reconstruction, and Probability Calculation (as shown in Figure 4-9). The system image exists since the system is defined. Its input range can be infinite except for undefined areas (e.g., $f(x) = 1/x$, $x = 0$ is undefined). The information generated by the Data Generation step provides the boundary for the system image.

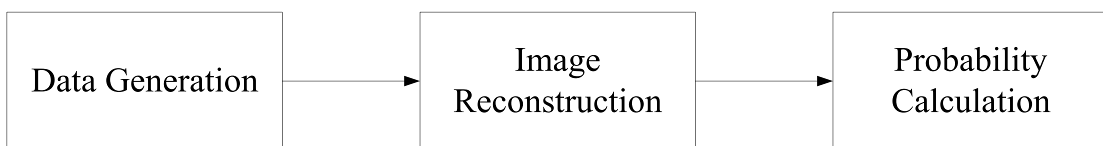

Figure 4-9 Image Reconstruction Method

"Data Generation" generates input data for "Image Reconstruction." The image of the system is reconstructed in Image Reconstruction. In practice, it is not necessary to reconstruct the whole image; instead, revealing the flat parts that contribute to the non-propagation is enough. "Probability Calculation" is used to calculate the non-propagation probability. In "Probability Calculation," the operational profile of the input variables and the flat parts found in "Image Reconstruction" are used in the propagation probability calculation.

4.4.2.1 Data Generation

The function of Data Generation is to generate a set of input data for Image Reconstruction {I_i, i = 1, 2…N, N is the number of the input vectors} according to limitations/dependencies among these input variables, where

I_i = {x_1, x_2,…,x_M}i, x_i is the ith input variable, i = 1, 2…, M. M is the number of input variables. The data on each dimension of input variables is ordered, i.e., $x_{i,k} < x_{i,k+1}$.

4.4.2.2 Image Reconstruction

Image Reconstruction is the most important step of the approach (that is why it is call Image Reconstruction method), because the accuracy of the method hangs on it. Fortunately, it is not necessary to explore the entire system image. Instead, only the flat parts are relevant. The data used for flat parts identification is generated in the Data Generation. The flat parts are identified by using the definition of flat parts in section 4.3.5. During identification, the dimensions of faulty variables are explored. It is possible that some part may be misidentified as flat part (as shown in Figure 4-10).

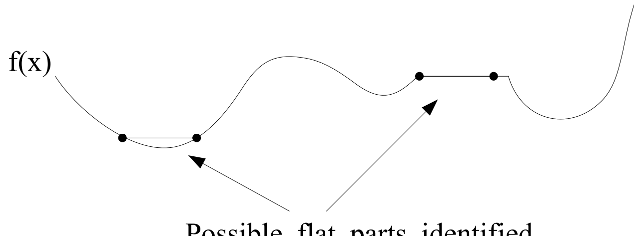

Figure 4-10 Flat Parts Identification

To prevent misidentification, the function value of one point (or more points, if required) randomly sampled in the possible flat part is calculated to confirm whether it is actually a flat part. The process of flat parts identification for a single

fault x is shown in Figure 4-11. The process for multiple faults is similar but more complex.

1. Generate the data $\{x_i, x_i \in [\min(x), \max(x)], x_i < x_{i+1}\}$.
2. If $f(x_i) = f(x_{i+1})$, then $[x_i, x_{i+1}]$ is considered a possible flat part.
3. Randomly select one or more points from the sub-interval, if they also generate the same output, then $[x_i + x_{i+1}]$ can be considered as one flat part.
4. Repeat 2-3 for all x_i, i=1,2,…,N. N is the number of the data

Figure 4-11 Algorithm of Flat Parts Identification for single fault

After the identification of flat parts, all the flat parts that have the same output are grouped as one PMA. Then the propagation probability can be calculated with (Eq. 4-1).

4.5 *Discussion of Image Reconstruction Method*

4.5.1 *Dependency Analysis*

It is the system itself that dictates whether a fault arising in the input is propagated to the output. The input range and the probability density distribution contribute to the non-propagation probability. In Image Reconstruction, it does not matter if the input variables are independent or not. The data provided for Image Reconstruction is only used to reveal the system image. Then, the dependency between variables is handled in Data Generation or Probability Calculation.

Non-propagation probability depends on the range of the flat parts and the probability density distributions of the input variables; the dependency of only these two properties of the input variables, Range Dependency and Probability Dependency, are considered.

Range dependency describes the limitation among the input variables. If two input variables, x_i and x_j, are independent, then for any given xi, xj has the same range distribution, and vice versa (see Figure 4-12).

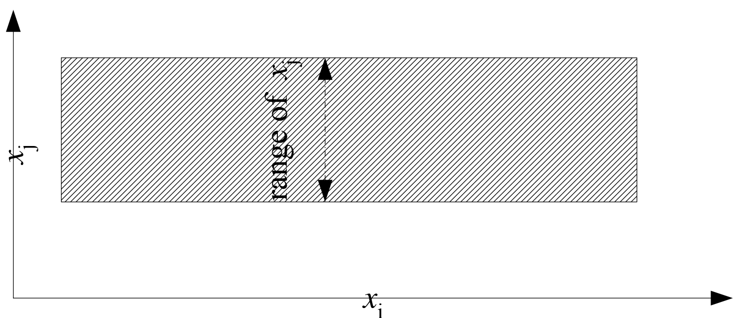

Figure 4-12 Independent Range distributions

If the range of one variable x_j is changing along with another variable x_i, then the range xi is dependent on x_j, i.e., the range distribution of x_i is a function of x_j (as shown in Figure 4-13). Sometimes, the range distribution of one variable is dependent on more than one variable. This will bring extra difficulty for the data generation.

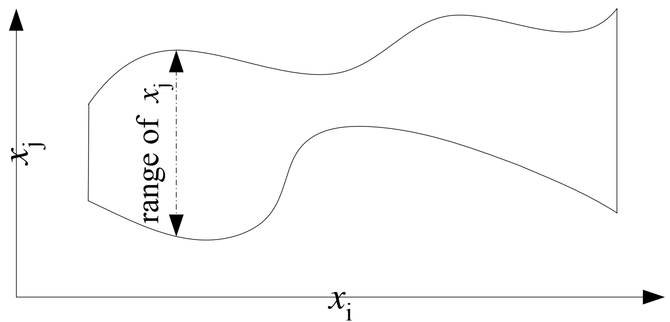

Figure 4-13 Dependent Range distributions

One special dependent case (as shown in Figure 4-14) shows that the range value of x_j is clearly defined as a function of xi, i.e., $x_j = f(x_i)$. In this case, the input vector $I \in R^n$ can be transformed to another input vector $I' \in R^{n-1}$ which has one less dimension through replacing x_j with $f(x_i)$.

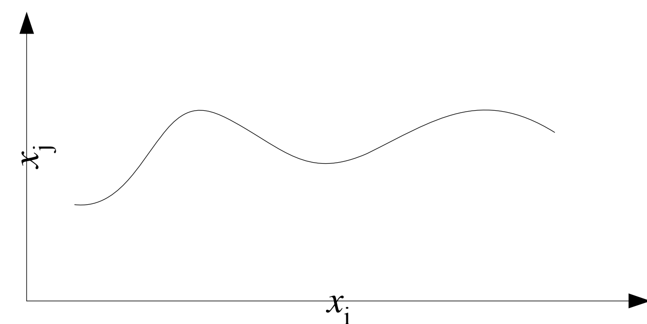

Figure 4-14 Dependent Range Distribution (special case)

Similar to the range dependency, the probability distribution of one input variable x_j may be dependent on another input variable x_i. When one variable x_j is range dependent on another variable x_i, it must be probability dependent on x_i, because the range is one parameter of the probability density distribution.

The existence of dependency between variables increases the complexity of the non-propagation analysis. The range dependency and probability dependency are handled in the Data Generation and Probability Calculation respectively.

If one of the input variables is incorrect, the variables that are range dependent on the variable must be incorrect, too. It means that a multiple input fault case has occurred. Data Generation generates data consistent with existing range dependencies and the data is used to identify the flat parts with regard to the dependent input variables.

The probability dependency is handled in Probability Calculation. If the input variables are independent, the expression of non-propagation probability for the multiple input faults. Using two input faults x and y as an example, we find:

$$prob = \int_{R_x} p_x(x)dx \int_{R_y} p_y(y)dy$$

where

R_x is the domain of x,

R_y is the domain of y.

If the input variables are dependent, the expression will be modified as:

$$prob = \int_{R_x} p_x(x)dx \int_{R_y(x)} p_y(x,y)dy$$

Where

$p(x,y)$ describes the probability dependency between x and y,

$R_y(x)$ is the domain of y which is a function of x.

4.5.2 Error Analysis

To estimate the relative error of the Image Reconstruction method, assume k flat parts exist in the software system (for simplicity, further assume they have the same length). Also, assume that both true input value and false input value are uniformly distributed in the domain. After dividing the input domain into N sub-intervals, the flat part c'd' is identified as shown in Figure 4-15 (the real flat part is cd).

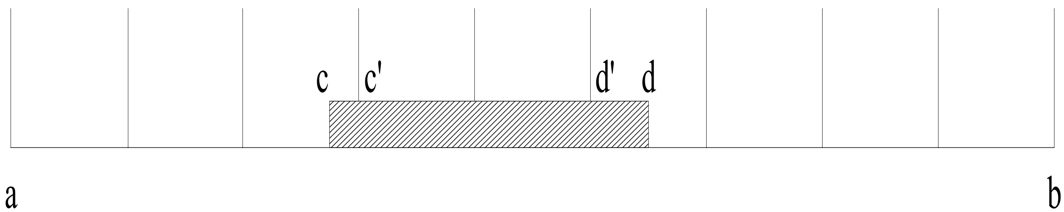

Figure 4-15 Error Analysis

Divide the input domain into N sub-intervals, the non-propagation probability in this iteration N is calculated as:

$$npp^{(N)} = k\left(\frac{\overline{c'd'}}{\overline{\overline{ab}}}\right)^2$$

And the real non-propagation probability is:

$$npp^* = k\left(\frac{\overline{cd}}{\overline{\overline{ab}}}\right)^2$$

Then the relative error $r^{(N)}$ is:

$$r^{(N)}=\left|\frac{npp^{*}-npp^{(N)}}{npp^{*}}\right|=\left|1-\frac{npp^{(N)}}{npp^{*}}\right|=\left|1-\frac{k\left(\frac{\overline{c'd'}}{\overline{ab}}\right)^{2}}{k\left(\frac{\overline{cd}}{\overline{ab}}\right)^{2}}\right|=\left|1-\left(\frac{\overline{c'd'}}{\overline{cd}}\right)^{2}\right|=\left|1-\left(\frac{\overline{cd}-\overline{cc'}-\overline{d'd}}{\overline{cd}}\right)^{2}\right|$$

$$=\left|1-\left(1-\frac{\overline{cc'}+\overline{d'd}}{\overline{cd}}\right)^{2}\right|\le\left|1-\left(1-\frac{2\frac{\overline{ab}}{N}}{\overline{cd}}\right)^{2}\right|=\left|\frac{4(\overline{ab})^{2}}{N^{2}(\overline{cd})^{2}}-\frac{4(\overline{ab})}{N\overline{cd}}\right|=4\left|\frac{1}{N^{2}\frac{npp^{*}}{k}}-\frac{1}{N\sqrt{\frac{npp^{*}}{k}}}\right|$$

Hence, the relative error with N sub-intervals is:

(Eq 4-3)

$$r^{(N)}=4\left|\frac{1}{N^{2}\frac{npp^{*}}{k}}-\frac{1}{N\sqrt{\frac{npp^{*}}{k}}}\right|$$

In the proof above, it was assumed $\overline{c'd'}$ had been identified, which means:

$\frac{\overline{ab}}{N}<\frac{\overline{cd}}{2}$ or

$N\frac{\overline{cd}}{2\overline{ab}}=\frac{N}{2}\sqrt{\frac{npp^{*}}{k}}>1.$

Then (Eq 4-3) becomes:

(Eq 4-4)

$$r^{(N)}=4\left(\frac{1}{N\sqrt{\frac{npp^{*}}{k}}}-\frac{1}{N^{2}\frac{npp^{*}}{k}}\right)$$

Where

npp* is the real non-propagation probability and k is the number of flat parts in the software.

Since npp* is unknown, it is impossible to estimate the relative error. However, the method is convergent because when the number of the interpolation

points used to reconstruct the system image goes to infinity, the approximation function will be exactly the original function and the error will be 0. In practice, reasonable npp and k are first assumed, and N is set to $N > \frac{2}{\sqrt{npp/k}}$. The method is then performed twice with N and 2N sub-intervals, respectively. If $|npp(2N) - npp(N)| < error$, npp(2N) can be seen as the propagation probability that can be used to assess the system reliability.

4.5.3 Efficiency Analysis

This method was applied to one 10-component system (see Figure 4-16) for single input fault and compared with random testing.

$$
\begin{aligned}
&Input\ (x, y)\ (x\ is\ the\ faulty\ input) \\
&component\ 1:\ \ x_0 = f_0(x, y) = x + y \\
&component\ 2:\ \ x_1 = f_1(x_0) = \begin{cases} 0 & 0 \le x_0 \le 1 \\ (x-1)^2 & otherwise \end{cases} \\
&component\ 3:\ \ x_2 = f_2(x_1) = \begin{cases} 2 & 0 \le x_1 \le 4 \\ x_1 - 2 & otherwise \end{cases} \\
&component\ 4:\ \ x_3 = f_3(x) = x^2 + 5x - 17 \\
&component\ 5:\ \ x_4 = f_4(y) = \sin(y) \\
&component\ 6:\ \ x_5 = f_5(x_1, x_3, x_4) = x_1 - x_3 x_4 \\
&component\ 7:\ \ x_6 = f_6(x_2, x_5) = |x_2 - x_5| \\
&component\ 8:\ \ x_7 = f_7(x, y) = x^2 \cos(y) \\
&component\ 9:\ \ x_8 = f_8(x_1, x_6) = x_1 \log(1 + x_6) \\
&component\ 10:\ \ x_9 = f_9(x_7, x_8) = \frac{x_7 - x_8}{x_7 + 1} \\
&Output\ x_9
\end{aligned}
$$

Figure 4-16 Example System

The experiment is designed in Microsoft Visual C++ 6.0 environment and executed on one computer with following configuration: CPU: PIII 500MHz, Memory: 256 MB RAM and OS: Windows XP (Professional Edition) with Service Pack 2. The results are shown in Table 4-2.

Table 4-2 Experiment Results

	Image Reconstruction Method		Random Testing	
Sample Size / #of interpolation points	Time Spent (sec)	Non-propagation probability	Time Spent (sec)	Non-propagation probability
100	0.701	0.000285	0.711	0.00034
250	1.502	0.00030875	1.743	0.000468
550	**3.715**	**0.00031910**	3.845	0.000369
600	4.166	0.00032175	4.156	0.000315
1000	7.25	0.00032835	7.611	0.000431

Our method's computation time is about the same as random testing for the same sample size/number of interpolation points. However, the result obtained from the Image Reconstruction method is more accurate than that from random testing for the same sample size/number of interpolation points.

It is shown [35] that the required sample size for random testing can be determined by (Eq. 4-5):

$$n = \frac{z^2 q}{r^2 p} \quad \text{(Eq. 4-5)}$$

Where

p is the estimated probability, q = 1-p.

r is the relative error

z is the upper (α/2)th point on the standard normal curve

For the 10-component system, the theoretical non-propagation probability is 1/3000. So, with α=0.05, r=0.1 (550 interpolation points are required in our method), the sample size for random testing is 1152480; then, the required computation time is $\frac{1152480}{1000} * 7.611 = 8771 >> 3.715$. Hence, the method is faster than random testing in this example. By solving equation (Eq 4-4), the number of interpolation points required by the method can be obtained as:

$$N = \frac{2\left(1+\sqrt{1-r}\right)}{r\sqrt{\frac{npp^*}{k}}} \quad \textbf{(Eq. 4-6)}$$

Hence,

$$\frac{N}{N_{Testing}} = \frac{\frac{2\left(1+\sqrt{1-r}\right)}{r\sqrt{\frac{npp^*}{k}}}}{\frac{z^2(1-npp^*)}{r^2 npp^*}} = \frac{2r\left(1+\sqrt{1-r}\right)\sqrt{knpp^*}}{z^2(1-npp^*)}$$

The value of $N/N_{Testing}$ for different npp* and k is listed in Table 4-3. Normally, the non-propagation probability is a very small number, therefore $N/N_{Testing}$ ≪ 1 and thus the Image Reconstruction method is more efficient than testing.

Table 4-3 $N/N_{testing}$ for different npp* and k (r=0.1, z=1.96)

npp \ k	1	2	5	10
0.0001	0.001	0.001	0.002	0.003
0.001	0.003	0.005	0.007	0.010
0.01	0.010	0.014	0.023	0.032
0.1	0.036	0.050	0.080	0.112

0.5	0.143	0.203	0.321	0.454
0.9	0.962	1.361	2.152	3.043
0.99	10.094	14.276	22.572	31.921

4.5.4 Scalability Study

As described in the previous sections, the method considers the remaining functions as one unit. Hence, it can be applied to any system no matter how large the system. The difference between the application to a small system and a large system is only the amount of computing time necessary to estimate the non-propagation probability.

The time (t) to estimate the non-propagation probability of a function f is related with the average execution time (τ) of every LOC, the length of the function (l), and the number of interpolation points (the number of sub-intervals) for all input variables (N). The relation can be expressed as:

$$t = Nl\tau \qquad \text{(Eq 4-7)}$$

For reference, we use the example system above (30 LOC) to estimate τ. In the example, the correct input variable y is divided into 1000 sub-intervals, and for every sample y, the incorrect input variable x is divided into N_x sub-intervals (N_x is expressed as (Eq. 4-6)).

Hence, the average execution time for every LOC can be obtained as:

$$\tau = \frac{3.715}{550*30*1000} = 2.25\times10^{-7}\ \sec/LOC = 2.25\times10^{-4}\ ms/LOC$$

Hence, equation (3) can be approximated as:

$$t = 2.25\times10^{-4}\,Nl$$

Table 4-4 shows the value of N for different r, npp* and k.

Table 4-4 Number of interpolation points required for different npp*, k, r=0.1

npp* \ k	1	2	5	10
0.0001	3897	5512	8714	12325
0.001	1232	1743	2756	3897
0.01	390	551	871	1232
0.1	123	174	276	390
0.2	87	123	195	276
0.5	55	78	123	174
0.9	41	58	92	130
0.99	39	55	88	124
0.999	39	55	87	123

Then, for a large system with 128k LOC, the calculation time can be estimated (as shown in Table 4-5).

Table 4-5 Calculation time (s) for different r, npp*, k, r=0.1 (l=128k)

npp* \ k	1	2	5	10
0.0001	112	159	251	354
0.001	35	50	79	112
0.01	11	16	25	35
0.1	4	5	8	11
0.2	3	4	7	8
0.5	2	2	4	5
0.9	1	2	3	4
0.99	1	2	3	4
0.999	1	2	3	4

4.5.5 Application for the different locations in Software

In the above discussion, it is assumed that the fault occurs at the input. For faults arising within the software, the method can be applied from the location of occurrence through the remaining functions. Establishing the non-propagation relationship between the input and output can be used for the non-propagation analysis of faults located anywhere in the software.

The software is divided into head function y = f(x) and tail function z = g(y) from the location y under study (as shown in Figure 4-17). Although the fault in y is propagated/masked in the tail function, some non-propagation information can be obtained through mapping x to y with the head function y = f(x).

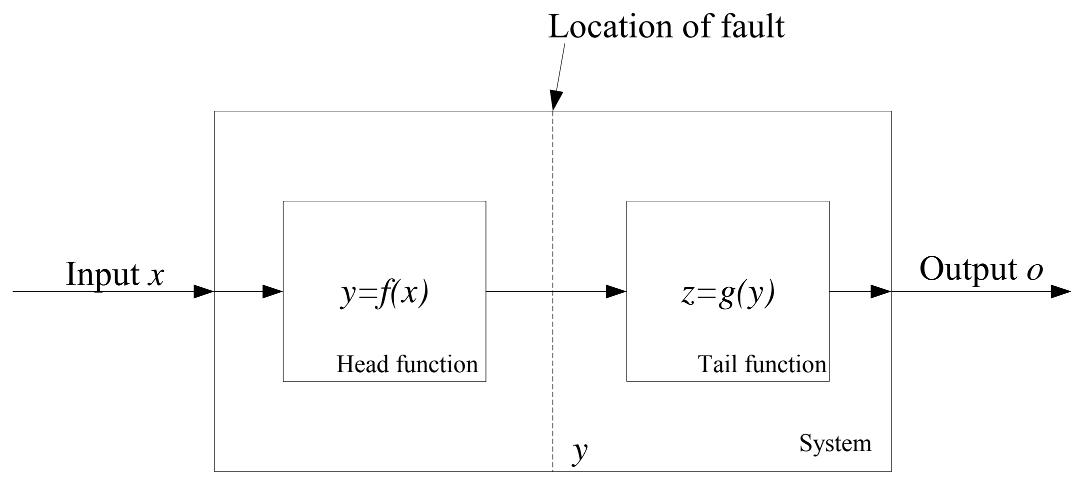

Figure 4-17 Non-propagation Analysis for a Fault located in the Software

As shown in Figure 4-18, let D_x represent the domain of *x*. The range of head function *y* = *f*(*x*) is D_y, $D_y = [\min(f(x)),\ \max(f(x)),\ x \in D_x]$. If L_x has been identified as the flat part with regard to x in D_x, i.e., $g(f(x)) = z_0, x \in L_x$ then, through the calculation in head function *f*(*x*), there must exist region L_y satisfying $L_y = [\min(f(x)),\ \max(f(x)),\ x \in L_x]$. Hence, Ly is a flat part with regard to y in D_y, i.e., $g(y) = z_0, y \in L_y$. If the value of the fault in location y and its true value belong to D_y, then, the non-propagation probability is $\left(L_y / D_y \right)^2$. For the region outside of D_y, the identification process has to be performed again, because the behavior in this region is still unknown.

This information is very significant. If the propagation analysis has to be performed for different locations, much unnecessary work will not need to be repeated and one can reduce the workload consequently. Such kind of information can not be obtained in random testing; therefore the same sample size in every location is required.

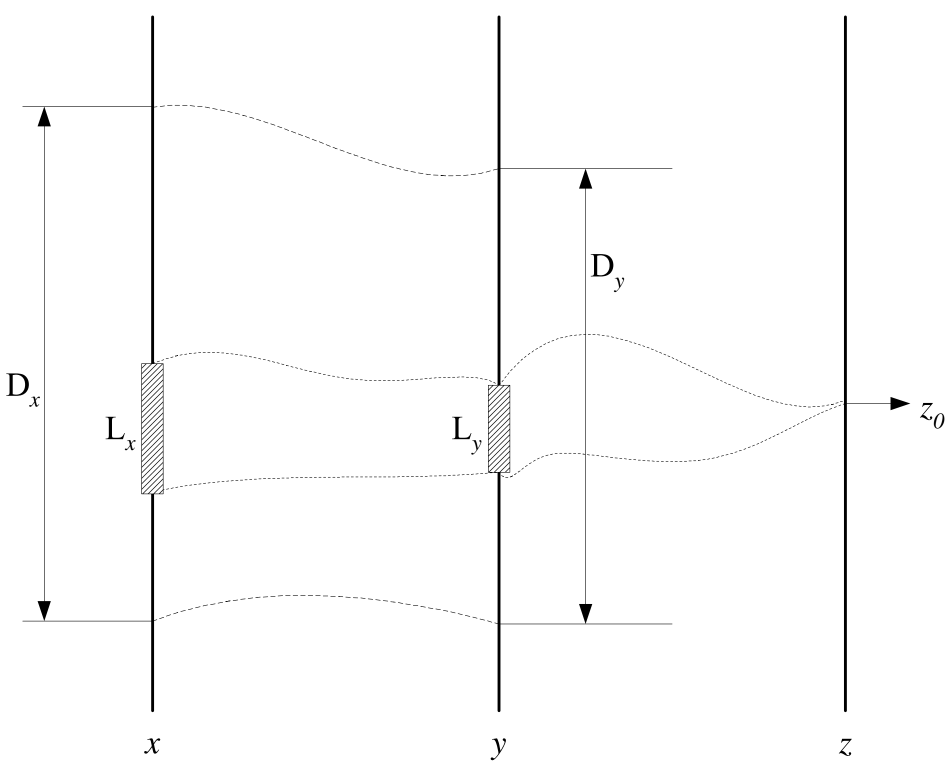

Figure 4-18 Mapping in the Head Function

5 Fault Propagation of other Value-Related Failure Modes

5.1 <u>Type Failure</u>

An incorrect input type is one of the input failure modes. For example, if the expected input type is integer, and the actual input type is *char*, then an incorrect input type failure occurs. Since the propagation is performed via the data state, how the input type failure causes the data state error is the key question for the propagation of incorrect input type failure (as shown in Figure 5-1).

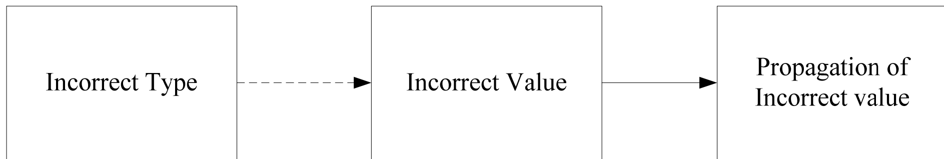

Figure 5-1 Transform form incorrect type failure to incorrect value failure

To answer this question, we need to know:

1) What are the data types supported by the computer system?

2) When is the date type checked?

3) What is the action taken by the computer when a type mismatch occurs?

5.1.1 *Data Type*

The *type* in computer science is a set of values and some operations which one can perform on the set of values. Programming languages implicitly or explicitly support one or more data types; these types may act as a statically or dynamically checked constraint on the programs that is valid in a given language.

Each programming language has a set of built-in primitive data types. At the same time, a language may also allow programmers to define new data types. The three basic elements of a specification of a data type are: attributes, values, and operations. The attributes distinguish data objects of the type; ANDthe values represent the possible values that can be taken by the data object of the type; the operations define the possible manipulations of the data objects of that type. For instance, a car data type in C++ is defined as:

```
Class car{
        int year;
        int maker;
        int price;
};
```

The attributes of the car data type are the dimensions (year, maker and price). The value of the data type is comprised of the sets of data that are valid values for the car data object, for instance, (1999, HONDA, 10000) and (1999, FORD, 11000) are the possible values for the car data type. The operations can "adjust the price."

The basic elements of the implementation of data type cover the storage representation and the manner of implementation. The storage representation represents the data objects of the data type in the storage of the computer during execution. The manner of implementation is particular algorithms or procedures that operate the chosen storage representation of the data type objects. There are three ways to implement the operation:

1. The operation is implemented as a hardware operation.
2. The operation is implemented as a procedure or function subprogram.
3. The operation is implemented as an in-line code sequence.

Here, only the primitive types are discussed, because they are common in almost all the languages. Typical primitive types may include: character, integer, floating-point number, Boolean, and reference. Different languages may have different primitive types.

In the earliest Intel processors, the floating point operations were not supported in hardware. This does not mean that these processors could not perform float operations. To perform float operations, the programmers had to use procedures composed of non-floating point instructions. Later, Intel provided an additional chip called a math coprocessor in which machine instructions were embedded so that floating point operations were performed much faster than when using a software procedure. The math coprocessor for the 8086/8088 was called 8087. The math coprocessors for 80286 and 80386 are 80287 and 80387, respectively. Since the Pentium generation, all 8086 processors have a built-in math coprocessor. Hence, floating operations are automatically supported.

5.1.2 Type Checking

Data storage representations that are built into the computer hardware usually include no type information, and the primitive operations on the data do no type checking. For example, a particular word in the computer memory during execution of a program may contain the bit sequence: 11100101100...0011. This bit sequence might represent an integer, a real number, a sequence of characters, or an instruction;

there is no way to tell. The hardware primitive operation for integer addition cannot check whether its two arguments represent integers; they are simply bit sequences, and the hardware operation must assume that they represent integers. A common programming error in assembly of machine languages is to invoke an operation such as integer addition on arguments of the wrong type. Such errors are particularly difficult to find because the operation does not fail in some obvious way. The operation "works," but the results produced are meaningless. However, subsequent operations may continue to compute with the "garbage" result to produce more "garbage," until the entire computation fails. Thus, at the hardware level, conventional computers are particularly unreliable in detecting data type errors.

"Type Checking" means checking that each operation executed by a program receives the proper number of arguments of the proper data type. Type checking typically occurs in 3 stages: coding, compilation, and execution (as shown in Figure 5-2).

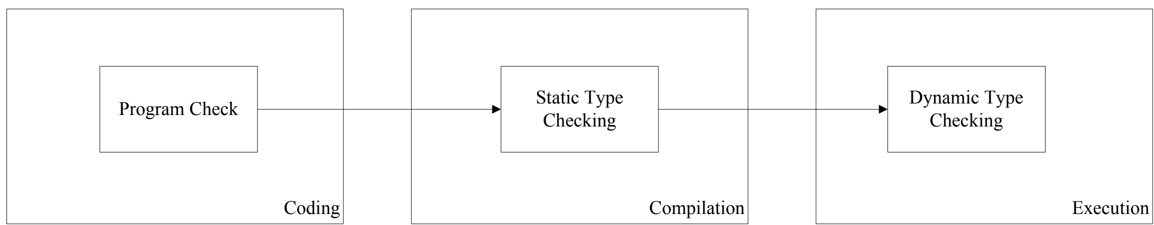

Figure 5-2 Type checking in different stages

5.1.2.1 Coding stage

During the coding stage, programmers may want to check the types of some special inputs; they will then check the type of the input with some type checking function available within the semantics and syntax of the language being used. For

example, in Visual Basic, in order to check whether a variant is numeric, the programmers may choose the following method (Figure 5-3).

```
'get variant v
If (IsNumeric(v)) Then
{'Continue Process}
Else
{'Error Handling}
End If
```

Figure 5-3 Example of Type Checking (Visual Basic)

When programmers use similar methods to check the type, they know how to handle type mismatches, or the requirements specifications address the problem explicitly. Thus, a failure due to type mismatches will not occur[3].

Programmers cannot check the types of all variables in the program. Most type checking takes place in the compilation stage via static type checking and in the execution stage via dynamic type checking.

5.1.2.2 Compilation stage

Most languages can check the data type (static type checking). Static type checking is performed in the compilation stage and it becomes a primary task of the semantic analysis carried out by a compiler. When translating a program, the compiler collects information from declarations in the program into various tables.

[3] On the other hand, error handling of type mismatches may directly impact the ability of an application to generate outputs within the time limit set by the system.

Then, the compiler checks each operation invoked by the program to determine if the type of each argument is valid. The information required for static type checking is:

1. For each operation, the number, order and data types of its arguments and results.
2. For each variable, the type of the data object named.
3. The type of each constant data object.

In static type checking, the statically typed languages and dynamically typed languages behave differently. Statically typed languages include C, C++, Java, ML, Haskell, etc., while dynamically typed languages include Objective-C, Scheme, Lisp, Smalltalk, Perl, PHP, Visual Basic, Ruby, Python, etc. For example, in the following pseudocode (Figure 5-4), the statically typed languages will report an error while the dynamically typed languages will allow it to execute.

```
var x;
x = 0;
x = “0”;
```

Figure 5-4 Statically vs. Dynamically Typed Variable

Static typing finds type errors reliably and at compile time. This should increase the reliability of the delivered program. Static typing usually results in compiled code that executes more quickly, because static type checking covers all the operations in the program statements, all possible execution paths are checked and therefore further testing for type errors is not necessary. When the compiler knows the exact data types that are in use, it can produce machine code that just does the right thing. Further, compilers in statically typed languages can find shortcuts more easily.

In static type checking, some extra codes are inserted into the compiled program to invoke the conversion. Usually, the statically typed languages prohibit the narrowing coercions because they save the time used to determine the validity of the coercion so that an efficient execution is obtained.

5.1.2.3 Executing stage

Static type checking cannot prevent all the type mismatches. For example, the user may provide a string instead of an integer through a man-machine interface. Hence, a dynamic type checking needs to be performed.

Dynamic type checking is run-time checking because variables can acquire different types depending on the execution path. It is performed right before the execution of an operation. In dynamic type checking, the type of a data object is indicated by storing a type tag. For example, the statement “double x=0.1;” defines a double data object which contains 0.1 as the double value and the “double” type tag. Then, in every operation, the computer checks the type tag of each argument first. If the argument types are correct, the operation is performed. Otherwise, an error is raised. After the execution of every operation, the appropriate type tag is also attached to the result.

We study the incorrect input type failure at the code level. In other words, the executable code is available which means that the code can be compiled and, therefore, type checking in the coding stage and the compilation stage has been completed. The incorrect input type thus occurs in the execution stage.

If the actual type of an argument is different from its expected type, then a type mismatch occurs. When a type mismatch is found, one of the two following actions is performed.

1. The type mismatch may be flagged as an error, and an appropriate error action taken.
2. A coercion (or implicit type conversion) may be applied to change the type of the actual argument to the correct type. The coercions are automatically invoked in some cases. For example, in C, for the operation: x*y, if x and y are type double and integer respectively, y is implicitly converted to type integer before multiplication is performed. Implicit type conversion, also known as coercion, is an automatic type conversion. Some languages allow, or even require compilers to provide coercion.

5.1.3 Type Conversion

Type conversion or typecasting refers to changing an entity of one data type into another. Most languages provide type conversions in two ways, implicit and explicit. In the case of implicit type conversion, the compiler is given the responsibility for determining that a conversion is required and how to perform the conversion. In the case of explicit type conversion, the programmer assumes the responsibility. An explicit conversion is applied to change the type of the actual argument to the correct type. The programmers invoke explicitly the conversions within a set of built-in functions. The programmers can cast an expression to coerce it to the correct type or the expected type. For example, (double) x can convert the value of x (type integer) to type double.

There are several kinds of explicit conversions:

- *Checked*. When the type mismatch occurs during execution, the program checks whether the destination type can actually hold the source value. If yes, the conversion is performed, otherwise, an error condition is raised.
- *Unchecked*. No check is performed before the conversion. If the destination type can not hold the source value, the result of conversion is undefined.
- *Bit Pattern*. The data is not interpreted at all and just the raw bit pattern is copied.

The basic principle of coercions is not to lose information. Some conversions (like from a short integer to a long integer) result in no loss of information. Such coercions are called widening or promotions. On the other hand, some coercions (like from double to long integer) may lose information. Such coercion is called narrowing. The coercion could be allowed if the data object has an appropriate value.

Different data types are stored differently as shown in Table 5-1.

Table 5-1 Storage of Data Type

Type	Size
char	1 byte
short integer	2 bytes
long integer	4 bytes
float	4 bytes
double	8 bytes

An integer can be directly expressed using bits as shown in Figure 5-5. The range of 16-bit and 32-bit integers are given by [-32768, 32767] and [-2147483648, 2147483647] respectively.

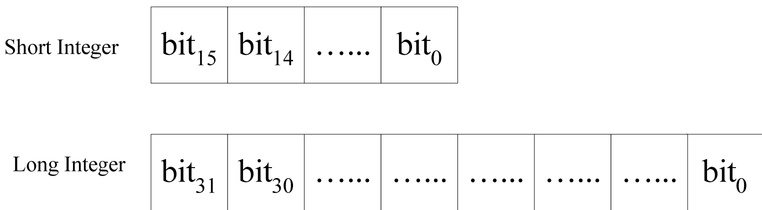

Figure 5-5 Representation of Integer

The representation of a floating point number is more complex. IEEE floating point numbers have three basic components: a *sign*, an *exponent*, and a *mantissa*. The mantissa is composed of a *fraction* and an implicit leading digit which is always equal to 1 and as such is never represented. The layout for single (32-bit) and double (64-bit) precision floating-point values is given in Figure 5-6.

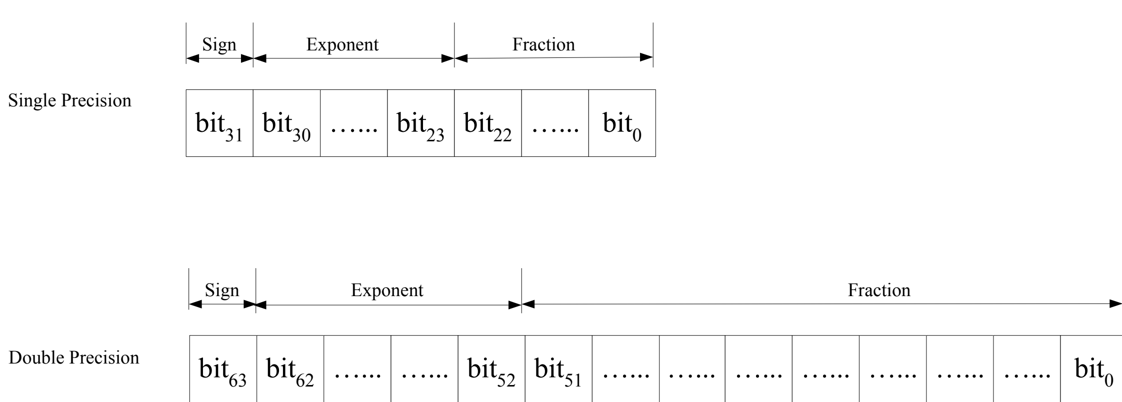

Figure 5-6 Floating Point Number Layout

The type conversion operations from one type to another type are summarized as Table 5-2.

Table 5-2 Type Conversion Operations

Source Type	Destination Type	Conversion
Char	Short Integer	Sign-extend.
Char	Long Integer	Sign-extend.
Char	Float	Sign-extend to long integer; convert long integer to float.

Char	Double	Sign-extend to long integer; convert long integer to double.
Short Integer	Long Integer	Sign-extend.
Short Integer	Float	Sign-extend; convert long integer to float.
Short Integer	Double	Sign-extend; convert long integer to double.
Short Integer	Char	Preserve low-order byte.
Long Integer	Float	Represent as float.
Long Integer	Double	Represent as double
Long Integer	Char	Preserve low-order byte.
Long Integer	Short Integer	Preserve low-order word.
Float	Double	Change internal representation.
Float	Char	Convert to long integer; convert long integer to char.
Float	Short Integer	Convert to long integer; convert long integer to short integer.
Float	Long Integer	Truncate at decimal point. If result is too large to be represented as long integer, result is undefined.
Double	Char	Convert to float; convert float char.
Double	Short Integer	Convert to float, convert float to short integer.
Double	Long Integer	Truncate at decimal points. If result is too large to be represented as long integer, result is undefined.
Double	Float	Represent as a float. If double value cannot be represented exactly as float, loss of precision occurs. If value is too large to be represented as float, the result is undefined.

The conversion may cause a loss of information. The loss for different conversions will be described in the following sections.

- **meaningless type change**

When a data object is converted from type char to numeric data types, it loses its property and produces “garbage” unless it is the programmers’ intention to perform these conversions based upon the contents of the requirements documents (in this case, there is no incorrect input type failure). The results in these conversions are not meaningful. Hence, we call the conversions from type char to numeric types or from numeric types to type char as meaningless type changes. Normally, these kinds of type conversions result in strange values of the destination type. The standard ASCII codes for the characters are shown in Table 5-3.

Table 5-3ASCII Code

Dec	Hex	Symbol	Dec	Hex	Symbol	Dec	Hex	Symbol
32	20	space	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	&	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	$	70	46	F	102	66	f
39	27	‘	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	140	78	x

57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	DEL

- **no loss of information**

  If the conversions are performed from type short integer to long integer, or float, or double, or from float to double, no loss of information occurs. In other words, the conversion is seamless.

- **Loss of information**

  Loss of information includes loss of precision, undefined value, and loss of magnitude.

  - **Loss of precision**

    If the conversions are performed from type float to long integer, or short integer, or from double to float, or long integer, or short integer, loss of precision may occur due to the truncation of the decimal fraction.

    According to Standard IEEE 754, the single precision floating number (float) has 8 significant figures and can be expressed as:

    $f = (-1)^s \times m_1.m_2m_3...m_8 \times 10^e$

    Where

    s is the sign bit,

e is the exponent,

m_1, m_2, … ,m_8 are the significant figures.

When a data object of other types (such as long and double) with more than 8 significant figures is converted to type float, some precision is lost. Similar to the float number expression, we express the data as:

$$d = (-1)^s \times m_1.m_2m_3...m_N \times 10^e$$

Where

N is the number of significant figures of the data.

Then the relative error due to conversion is:

$$relative\ error = \left|\frac{d-f}{d}\right| = \left|\frac{(-1)^s \times m_1.m_2m_3...m_N \times 10^e - (-1)^s \times m_1.m_2m_3...m_8 \times 10^e}{(-1)^s \times m_1.m_2m_3...m_N \times 10^e}\right|$$
$$= \left|\frac{0.0000000m_9m_{10}...m_N}{m_1.m_2m_3...m_N}\right| < \left|\frac{0.0000001}{m_1.m_2m_3...m_N}\right| \le 10^{-7}$$

When the data type is converted from a single precision floating point number to an integer, the floating point number is truncated at the decimal point, and the maximum absolute error is 1.

- **Undefined value**

When the conversions are performed from float to long integer or short integer; or from double to long integer or short integer, if the value is too large to be represented as the destination type, the

results are different in different languages. For instance, in C++ and PHP, the result is a numeric value which has no relation with the initial value, while in JAVA, the result is the maximum number which can be represented by the destination type. More precisely, in JAVA, when converting a single precision floating point number to an integer, if the floating point number is greater than 2147483647, then the conversion result will be 2147483647. If the float pointing number is less than -2147483648, then the conversion result will be always -2147483648.

- **Loss of Magnitude**

When the conversions are performed from long integer to short integer, only the lower-order word is preserved. The higher-order word is truncated. The long integer can be expressed in base 2 as:

$$l = \left(b_{31}b_{30}......b_0\right)_2$$

After the conversion, the short integer is:

$$s = \left(b_{15}b_{14}......b_0\right)_2$$

Then the relative error is:

$$\left|\frac{l-s}{l}\right| = \left|\frac{(-1)^{b_{31}} \times \left(b_{30}b_{29}......b_0\right)_2 - (-1)^{b_{15}} \times \left(b_{14}b_{13}......b_0\right)_2}{(-1)^{b_{31}} \times \left(b_{30}b_{29}......b_0\right)_2}\right|$$

$$\leq \left|\frac{\left(b_{30}b_{29}......b_0\right)_2 + \left(b_{14}b_{13}......b_0\right)_2}{\left(b_{30}b_{29}......b_0\right)_2}\right| \leq 2$$

And the absolute error is:

$$|l-s| = \left|(-1)^{b_{31}} \times (b_{30}b_{29}......b_0)_2 - (-1)^{b_{15}} \times (b_{14}b_{13}......b_0)_2\right|$$
$$\leq \left|(b_{30}b_{29}......b_0)_2 + (b_{14}b_{13}......b_0)_2\right| \leq (2^{31}-1)+(2^{15}-1)$$
$$= 2147516414$$

$$|l-s| = \left|(-1)^{b_{31}} \times (b_{30}b_{29}......b_0)_2 - (-1)^{b_{15}} \times (b_{14}b_{13}......b_0)_2\right|$$
$$\geq \left|(b_{30}b_{29}......b_0)_2 - (b_{14}b_{13}......b_0)_2\right| = (b_{30}b_{29}......b_{15}000000000000000)_2$$

The result of conversion is shown in Figure 5-7.

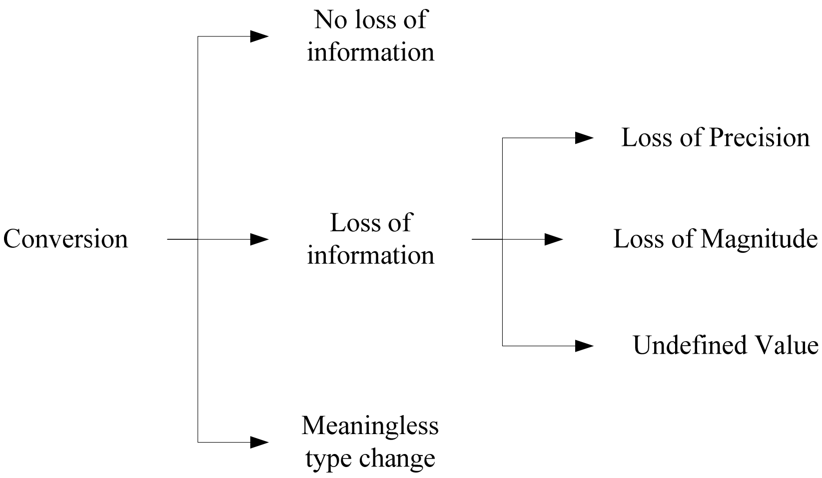

Figure 5-7 Result of Conversion

5.1.4 Summary

Based on the discussion above, the possible results for different conversions are shown in Table 5-4.

Table 5-4 Results for different conversions

Source Type	Destination Type	Conversion
Char	Short Integer	meaningless type change
Char	Long Integer	meaningless type change
Char	Float	meaningless type change
Char	Double	meaningless type change
Short Integer	Long Integer	No information loss

Short Integer	Float	No information loss
Short Integer	Double	No information loss
Short Integer	Char	meaningless type change
Long Integer	Float	Loss of precision
Long Integer	Double	Loss of precision
Long Integer	Char	meaningless type change
Long Integer	Short Integer	Loss of magnitude
Float	Double	No information loss.
Float	Char	meaningless type change
Float	Short Integer	Loss of precision; undefined value; Loss of magnitude
Float	Long Integer	Loss of precision; undefined value; Loss of magnitude
Double	Char	meaningless type change
Double	Short Integer	loss of precision; undefined value; Loss of magnitude
Double	Long Integer	Loss of precision; undefined value
Double	Float	Loss of precision; undefined value

As discussed in the above sections, the propagation of incorrect type failure can be summarized as shown in Figure 5-8. Now, the propagation probability of incorrect input type failure can be estimated with the Image Reconstruction Method.

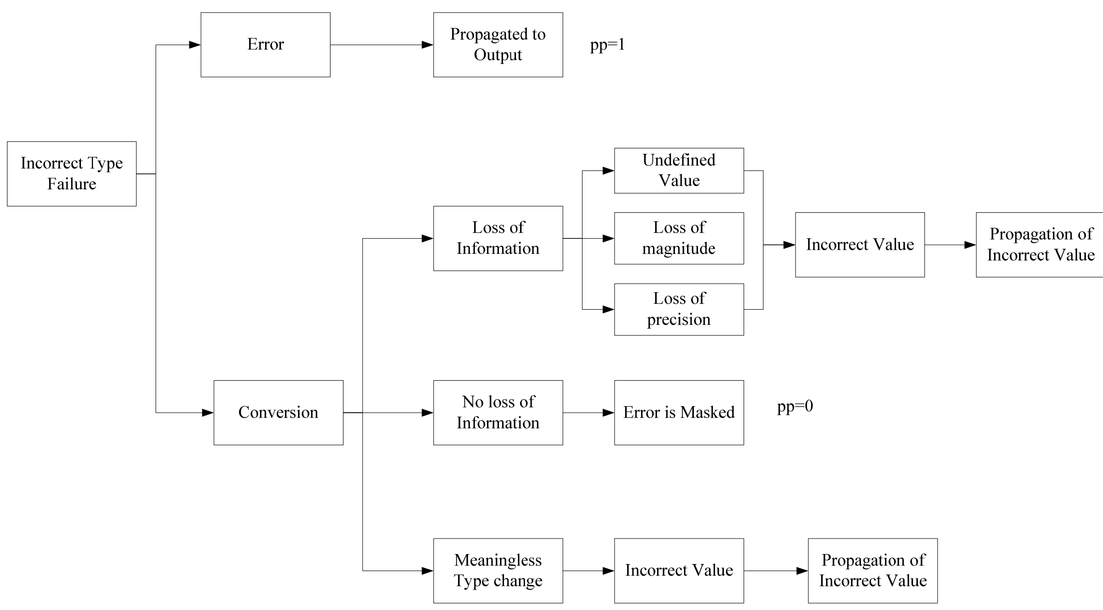

Figure 5-8 Propagation of Incorrect Type

5.2 *Range Failure*

5.2.1 *Definition*

The range defines the values a quantity may take. In [2, 3], it is formally defined as:

$$Rg(I) = \left[\min\left(V(I)\right), \max\left(V(I)\right)\right]$$

V(I) represents the value of the input.

One must note that the range may consist of only one region, e.g., Rg=[0,100], or it may consist of several regions, e.g., Rg=[0,13] $\cup$ [50,79] $\cup$ [91,100]. For the sake of simplicity, we will use $\left[\min\left(V(I)\right), \max\left(V(I)\right)\right]$ to represent the range which could be composed of the union of a set of regions.

5.2.2 *Out of Range Failure*

The possible failure mode is "out of range", i.e., $V(I) \notin Rg(I_e)$. For example, if the expected range is $Rg(I) = [0,100]$ and the input value is 110, then an out of range failure occurs. As a special case of value failure, the out of range value is assigned to its variable directly and causes a data state error.

Essentially, the computer's memory is made up of bytes. Each byte has a number and an address associated with it. As soon as a variable is declared, the amount of memory needed is assigned for it at a specific location in memory (its memory address). For example, as shown in Figure 5-9, an integer variable has a value 25. The variable is stored in memory location 1800.

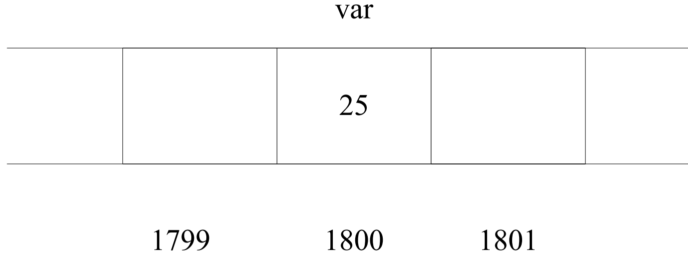

Figure 5-9 Variable and its address

Although we generally do not actively decide the exact location of the variable and just operate on the value of the variable, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to perform some operations related with the positions.

The variables can be divided into two categories: address-related and non address-related. The failures in the address-related variables are critical for the system. Kao, Tang and Iver [27] found that pointer faults tend to crash the system

immediately if those faults are activated. The pointer faults they used in their experiment are the same as the address-related failures introduced in this dissertation.

It is easy to explain their experimental results. If the address value is not correct, the computer will access a different memory address. The wrong memory address may be prohibited for accessing. For example, the attempt to read data from an array, A(10) (its range is $[0,9]$), by using A[-1] will be reported as an error.

The system will likely crash even when the action is allowed to happen. If the software performs a write action on the wrong address, it will change the value at that address instead of the expected address. Hence, it may cause a value failure to the software because the value at the expected address is not updated as expected. At the same time, updating a value at a wrong address will cause some unpredictable failure to other running applications or even to the operating system. If the value is used by another application, it may cause this application to fail. If the failure in other applications further impacts the operating system, the software component being studied will fail sooner or later due to the operating system failure. If the failures in other applications have no impact on the operating system, the software will not be affected. Because our target system includes only the operating system and the software under study and the failures in other applications do not impact our target system, so the failure can not be detected in our system.

If the value is used by the operating system, it may cause a crash of the operating system. Although it is possible that the write action does not affect the operating system, the probability is very small. So, we would rather claim that the operating system will crash and the software fails consequently.

If the software just performs a read action on the wrong address (if it could), it may read incorrect values and the out of range failure is transformed to the value failure. Then, its propagation criterion can be therefore expressed as $f\left(I_e^t\right) \neq f\left(I_f^t\right)$.

If the out of range value is non address-related, it will just impact the value of the variable. The out of range value may be replaced by a boundary value. For example, if the input to an integer is greater than 2147483647, then the number 2147483647 may be assigned to the variable. Let us denote by I_{bL} and I_{bU} the values of the lower and upper bound of the expected range respectively. Then the propagation criterion for this case can be expressed as:

$$\begin{cases} f\left(I_e^t\right) \neq f\left(I_{bL}\right), \text{ if } I_f^t < I_{bL} \\ f\left(I_e^t\right) \neq f\left(I_{bU}\right), \text{ if } I_f^t > I_{bU} \end{cases} \quad \textbf{(Eq 5-1)}$$

The out-of-range value may be out of the range that the computer can represent (physical range) when the out-of-range value is not automatically adjusted. This may cause an undefined value and probably cause a crash of the software. If the out-of-range value does not exceed the physical range, it will cause a value failure and the propagation criterion is the same as the value failure, i.e., $f\left(I_e^t\right) \neq f\left(I_f^t\right)$.

By summarizing the discussion above, the propagation of the out-of-range failure is described in Figure 5-10.

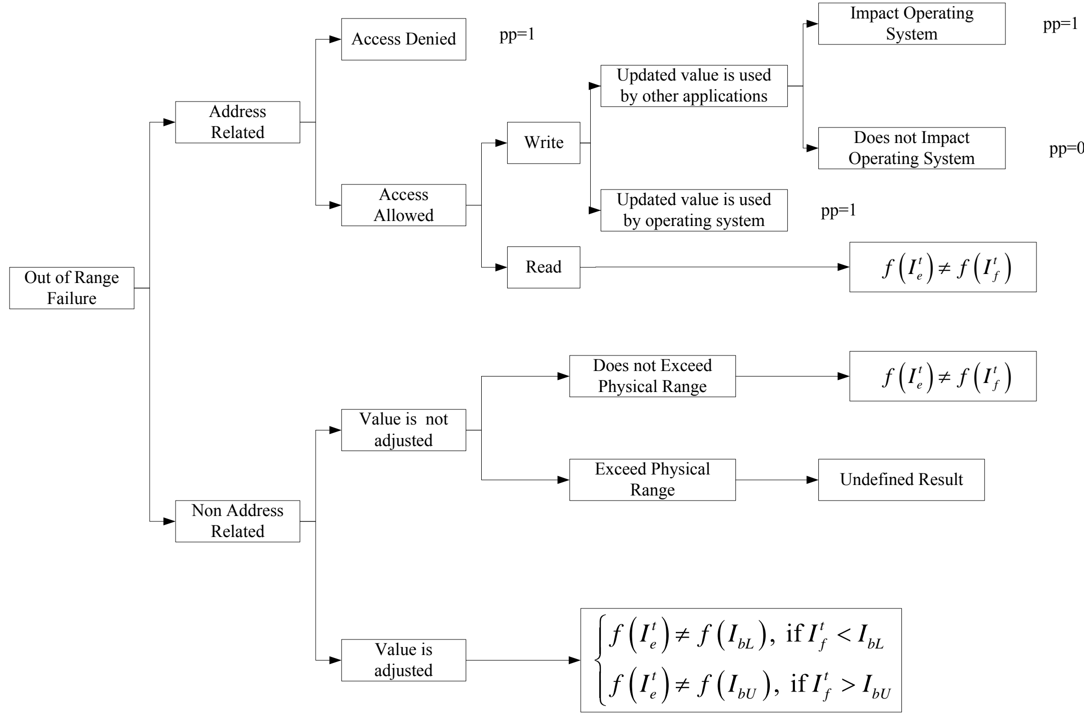

Figure 5-10 Propagation of Out of Range Failure

5.3 *Amount Failure*

5.3.1 *Definition*

The amount is the total number or quantity of input, denoted by A(I). A(I) is defined as: $A(I)=|I|$, where "$|I|$" yields the number of elements in the vector I.

The possible failure modes are "too much" and "too little" amount of input. They are defined as:

Too much: $A(I)>\upsilon$

Too little: $A(I)<\upsilon$

Where

υ is the number of elements in the expected input.

For example, in PACS, the user is expected to provide the SSN of an entrant and his/her Last Name at the same time. Then the amount of input is $|SSN, LastName| = 2$. If the user only provides SSN or LastName, then a too little amount of input failure occurs. On the other hand, if the user provides SSN, LastName and PIN, then a too much amount of input failure occurs.

5.3.2 Propagation of Amount Failure

Let us denote the n-element-input by $I = \{v_1, v_2, ..., v_n\}$. The too little amount failure means that a sub-set of the elements is missing. The elements in an input vector may come from different components or they may come from the same component.

For the too little amount failure, without losing generality, let us assume that the actual amount is n', $n' < n$; and the ith element is missing. If the elements come from different components, which means some or all of the components fail to send information to the software component, the absence will cause a "too late" failure which will be described in 6.2.3.

If the elements come from the same component, missing an element will create a shift in the position of the elements following it. The shift will cause value-related failures to the elements $v_i, v_{i+1}, ...,$ and v_n. The actual input can be expressed as $I = \{v_{e,1}, v_{e,2}, ..., v_{e,i-1}, v_{f,i}, v_{f,i+1,} ......, v_{f,n}\}$. From the expression, we can see that the first i-1 elements are the same as the expected ones while the remaining elements may be different from their expected values. Because the elements v_i, v_{i+1}, …, and v_n may have different type and range, the difference may cause value, range, or type failure.

For instance, if the type of $v_{f,i}$ is different from the type of $v_{e,i}$, a type failure occurs. Furthermore, some elements, v_n, v_{n-1}, etc., may be assigned to an empty value, that makes it appear like the elements have not been provided. For instance, if the length of v_i is equal to or larger than the summation of the length of the remaining elements, the remaining elements will be treated as element v_i and there are no inputs provided to the remaining elements. Then, it seems that the remaining elements are missing. The propagation analysis for the omitted failure is described in section 6.2.3. This is a case where multiple omissions may occur. Denote the failure in the element by $v_{f,j}, j = i, i+1, ..., n$. The propagation criterion for this situation can be expressed as:

$$f\left(\left\{v_{e,1}, v_{e,2}, ..., v_{e,n}\right\}\right) \neq f\left(\left\{v_{e,1}, ..., v_{e,i-1}, v_{f,i}, v_{f,i+1}, ..., v_{f,n}\right\}\right) \quad \textbf{(Eq. 5-2)}$$

The propagation of "too little" amount failure can be described as Figure 5-11.

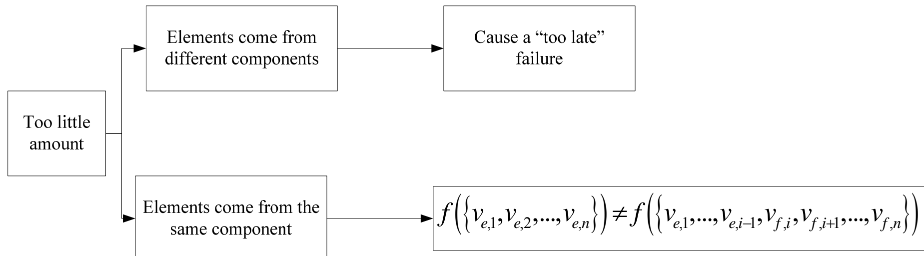

Figure 5-11 Propagation of Too Little Amount Failure

The too much amount failure means that one or some additional elements are provided along with the n required elements, i.e., $I = \left\{v_1, v_2, ..., v_n, v_{n+1}, ..., v_{n+k}\right\}$, $k \geq 1$. The propagation behavior depends on how these additional elements are handled.

If the elements come from different components, the additional elements are not accepted, because the software only gets inputs from the expected components. Hence, the additional elements do not impact the software.

When the elements come from the same component, if the elements are provided and read in fixed formats and the additional elements are provided after the n expected elements, the n elements will be read correctly and no additional elements are read. Hence, the too much amount failure does not propagate. For example, if the software reads one character for each element, then only n characters are read for the n expected elements. If the additional elements are stored in a buffer, it will be used as next input and may cause an input failure to next input. If the additional elements are inserted in the middle of the n expected elements (assume that they are inserted after the ith expected element without losing generality), it may cause other value-related (value, type or range) failure to the remaining expected elements. If the type of the type of the additional element is different with the type of the expected element, a type failure occurs. If the value of the additional element is different with the value of the expected element, a value failure occurs. If the value exceeds the range of the expected range or physical range, a range failure occurs.

If the software does not read the elements in fixed formats, the additional elements will be treated as part of the nth element and cause a value-related (value or range) failure to it. If the additional elements are inserted in the middle of the n expected elements, it may cause other value-related failure to the elements as discussed in the above. Then, the propagation of the too much amount of input failure can be summarized as Figure 5-12.

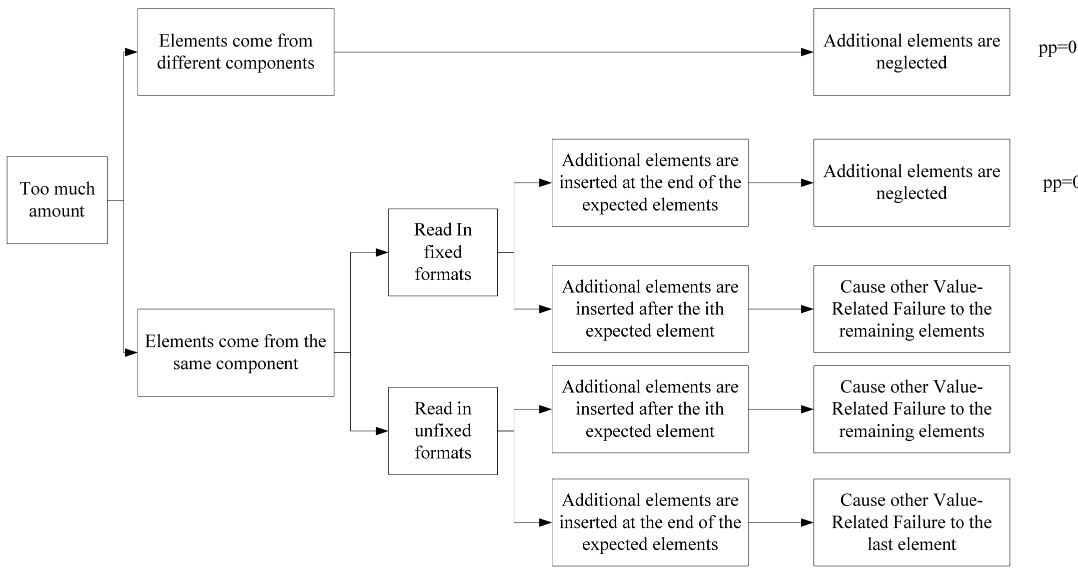

Figure 5-12 Propagation of too much amount failure

6 Propagation of Time-Related Failure Modes

6.1 *Time-Related Failure Modes*

Time plays a critical role in many modern software systems, especially in real-time systems. Time-related failure modes cover the characteristics of the input such as time, rate, and duration. In the following section, the propagations of these failure modes are discussed.

6.2 *Timing Failure*

6.2.1 *Definition*

The software input usually functions within a time window $[t_a, t_b]$. We assume that both software and system requirements share the same time window specifications. Some software systems are sensitive to the length of the time window. If the input arrives before the time window (i.e., $t < t_a$) a premature (too early) failure occurs. If the information is provided after the time window (i.e., $t > t_b$) a delayed (too late) failure occurs. If the input arrival time t goes to infinite, an omitted failure mode occurs. Obviously, the omitted failure mode is a special case of delayed failure mode. This is illustrated in Figure 6-1.

Figure 6-1 Time Window for Input

6.2.2 Computational Model

To analyze the behavior of timing failures, the software can be modeled as the combination of (i) a data processing unit, (ii) an input buffer (of size n) and (iii) a time window (of duration d) that appears repeatedly (i times) during the software operation with a separation s between two consecutive time windows. This is illustrated in Figure 6-2. Input data is accepted only while a time window is activated. Only one input (which may consist of a complex set of values) is accepted per time window. If the input data arrives out of a time window and the buffer size n is equal to 0 (i.e., the buffer is full or there is no buffer) the input data will be lost. Otherwise, if n is greater than zero, the input will be stored in the buffer and accepted when the next window is activated. Note that by varying the specifications of n, d, i and s, the proposed model can cover all the different ways a software system deals with input data.

The propagation of timing failures depends on whether or not the data processing unit implements fault tolerance strategies (i.e., error detection and recovery mechanisms) that can deal with the error. Accordingly, three possible reactions can be foreseen for the data processing unit if a timing failure exists:

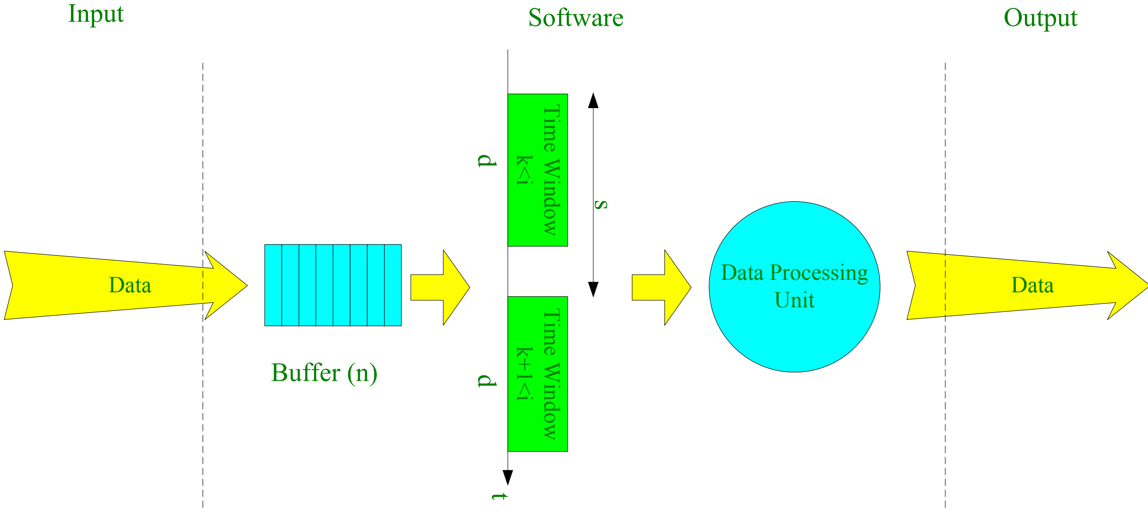

Figure 6-2 Computational Model for timing failure

Case 1. No fault tolerance mechanism is available. The timing failure thus propagates to the output (e.g., in an image processing software, a missing data packet will lead to a pixel not being displayed on a screen).

Case 2. Error detection mechanisms (e.g., a watchdog timer) exist and implement safety strategies, such as stopping or crashing the software system.

Case 3. The data processing unit provides error recovery mechanisms (with or without associated detection mechanisms) that can handle the situation (e.g., an exception handler that triggers a forward recovery action). In this case, a recovery action may successfully mask the timing failure.

In the following sections, the propagation of the too late, omitted, and too early timing failures are analyzed according to the behavioral framework introduced above.

6.2.3 Too Late (and Omitted) Failure Mode

The too late failure (also called delayed failure, timeout failure or deadline missed) occurs when the input data arrives after its corresponding time window. The three possible reactions of the data processing unit to this situation are:

Case 1. No fault-tolerance mechanisms available. In this case, the timeout failure does not affect the data state directly. Instead, it only affects the time needed to generate the output, i.e., the output is delayed or omitted. Since the data state is not infected, the Image Reconstruction Method is not applicable. Note that the delayed or omitted software output may not be considered a failure from the system viewpoint if the system requirements tolerate the occurrence of software delays (e.g., as is the case for soft real-time systems). The propagation criterion from the system viewpoint can be expressed as follows:

$$t_o = t_a + \tau > t_o^{\text{limit}}$$

where:

t_o is the time at which the output is generated,

t_i is the time at which the input arrives (which can be ∞ in case the input is omitted),

τ is the minimum execution time required to produce an output,

t_o^{limit} is the time limit specified at system level before which the software output should be provided.

Case 2. Error-detection mechanisms available. The error-detection mechanisms (e.g., exception handlers, watchdog timers, etc.) provided by the software may detect the timing failure and signal it. Usually, such a detection is followed by the execution of a safety action – stopping or crashing the software. We

consider this situation as a kind of fault propagation. Hence, we assume the propagation probability is 1 when the timing failure is detected. This case is covered by the Image Reconstruction Method since the missing input can be represented by a NULL value (i.e., the void set). It is reasonable to do that since the data processing unit does not receive any data during the time window. The software output can also be defined as a NULL. This can be formally expressed as follows:

$$f(NULL) = NULL \neq f(I_e) .$$

When a timing failure is not detected by the error mechanisms, the situation is the same as the one described in Case 1 above.

Case 3. Recovery mechanisms available. This case represents the situation in which the software provides recovery mechanisms that can mask the timing failure. For example, when a too-late failure occurs, the data processing unit may get some predefined data from memory and use it as an input to continue the computation. The predefined data may be a constant value or a dynamic value.

If the predefined data is a constant value, c, the propagation criteria can be expressed as:

$$f(I_e) = f(I_r) = f(c)$$

Then, only the PMA that maps to $f(c)$ is required to calculate the fault propagation probability. The Image Reconstruction Method presented in 4.4.2 can be simplified as follows:

1. Divide the input domain, $[I_{min}, I_{max}]$ into N sub-intervals with N+1 interpolation points:

   $\{I_i, i=0, 1, 2, \ldots, N$ and $I_{min} = I_0 < I_1 < I_2 < \ldots < I_N = I_{max}\}$.

2. Calculate function f using the interpolation points. If two consecutive interpolation points, I_i and I_{i+1}, generate the same output and such an output is equal to $f(c)$ (i.e., $f(I_i) = f(I_{i+1}) = f(c)$) then randomly select one or more points from sub-interval $[I_i, I_{i+1}]$. If these latter points are equal to $f(c)$, then $[I_i, I_{i+1}]$ can be considered as one flat part which can potentially mask the too late failure.

3. Group all the flat parts identified in step 2 as one PMA.

Figure 6-3 Simplified Algorithm

Once the PMA is identified, the propagation probability for the timeout failure can be estimated using (Eq. 4-1).

On the other hand, the predefined data may not be a fixed value, but rather a dynamic value stored in memory that is periodically updated by the software system. Let m represent the dynamic data stored in memory. Then, for any given m, the probability that the fault is masked corresponds to the probability that the output related to the expected input, I_e, is equal to the output related to the predefined data m. In general, the propagation probability for all possible predefined data can be expressed as:

$pp = 1 - \int p(m) p[f(I_e) = f(m)] dm$ $= \int p(m) \{1 - p[f(I_e) = f(m)]\} dm$	**(Eq 6-1)**
where: $p(m)$ represents the probability density function of value m, $p\left[f(I_e) = f(m)\right]$ represents the probability that the output related to the expected input is equal to the output generated by value m.	

In practice, the propagation probability for predefined data can be calculated by combining the algorithm in Figure 6-3 and a numerical integration method. The corresponding algorithm is shown in

Figure 6-4.

Probability $pp(i)$ in step 6 represents the contribution of the ith interval $\left[m_{i-1}, m_i\right]$ from the probability density function $p(m)$ to the propagation probability of value m_i. Since value m may vary, the average value $\frac{\left[p\left(m_{i-1}\right) + p\left(m_i\right)\right]}{2}$ is used to represent the probability density in such an interval.

The propagation of delayed failure mode is summarized in Figure 6-5

1. Determine the range of predefined data *m*, *R*(*m*), and its probability density function, *p*(*m*). Values *m* and *p(m)* can be obtained from the software operational profile.
2. Divide *R*(*m*) into *k* sub-intervals with *k*+1 interpolation points:

   $\{m_i, i=0, 1, 2, ..., k \text{ and } m_{min} = m_0 < m_1 < m_2 < ... < m_k = m_{max}.\}$
3. Set *i*=1.
4. Use the algorithm in Figure 6-3 to identify the PMA that maps to $f(m_i)$.
5. Using (Eq 6-1), calculate the propagation probability for value m_i, $pp(m_i)$.
6. Calculate $pp(i) = pp(m_i)[p(m_{i-1}) + p(m_i)]/2$.
7. Increase *i* by 1.
8. Repeat steps 4 ~ 6 until *i* is greater than *k*.
9. Calculate the final propagation probability, $pp = \sum_{i=1}^{k} pp(i)$.

Figure 6-4 Algorithm to Calculate Probability for predefined data

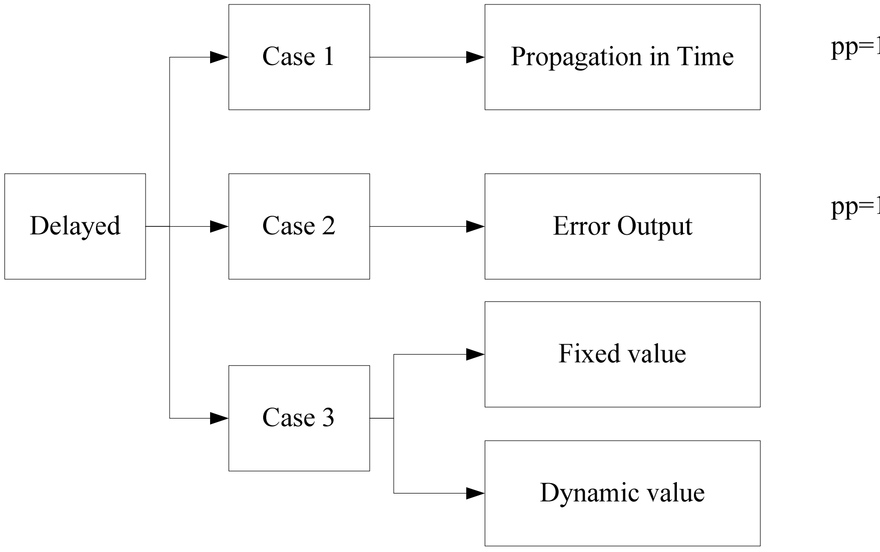

Figure 6-5 Propagation of Delayed Failure Mode

6.2.4 Too Early Failure Mode

Too-early (or premature) failures occur when the input arrives before the time window. The three possible reactions of the data processing unit to this situation are:

Case 1. *No fault tolerance mechanisms available*. If the input arrives before the time window and the software is not ready to receive it (i.e., the available buffer size is 0), the input will be lost and no output will be generated. In this case, the too early failure is propagated. As described in the Case 1 in section 6.2.3, the omitted software output may not be considered a failure from the system viewpoint if the system requirements tolerate software delays.

On the other hand, if the software accepts the input (i.e., the early input is initially stored in the buffer and used at the beginning of the time window) a correct output will be produced, both in the value and time domains. In this case, the propagation probability is 0. This situation can also be seen as a masking of the failure due to the intrinsic fault tolerance properties of the software algorithm.

Case 2. *Error detection mechanisms available*. This situation is similar to Case 2 of Section 6.2.3.

Case 3. *Recovery mechanisms available*. In this case, several types of recovery mechanisms might mask the too early failure. For example, an explicit recovery mechanism (similar to a buffer) could be implemented that allows for storing the early input and using it when the time window is activated. This situation is similar to Case 1 described above. The propagation probability will thus depend on the efficacy of the recovery mechanisms to mask the too-early failure.

6.3 Rate Failure

6.3.1 Definition

Rate of the input is the frequency at which the input is received. It is formally defined in [1, 2] as:

$$R(I_i) = \frac{m}{T_{j+m}(I_i) - T_j(I_i)}$$

Where:

T_j is the time of jth occurrence of I_i,

T_{j+m} is the time of $(j+m)^{th}$ occurrence of I_i,

m is the number of occurrences of Ii between T_j and T_{j+m}.

Two necessary conditions are required to guarantee that the rate is one of the characteristics of the input to the software component.

1) The inputs are sent to the same receiver in the software component. More precisely, the same variable is sent to the I/O portion of the software component. The concept of rate is meaningless when inputs address different variables.
2) The input appears periodically.

Normally, there exists a range for the rate: $[\upsilon_L, \upsilon_U]$, where υ_U is the expected upper bound of the rate and υ_L is the expected lower bound of the rate. Therefore, the "too fast" and "too slow" rate failure modes can be defined as:

Too fast:, $R(I_i) > \upsilon_U$

Too slow: $R(I_i) < \upsilon_L$

Figure 6-6 illustrates too fast and too slow failure modes.

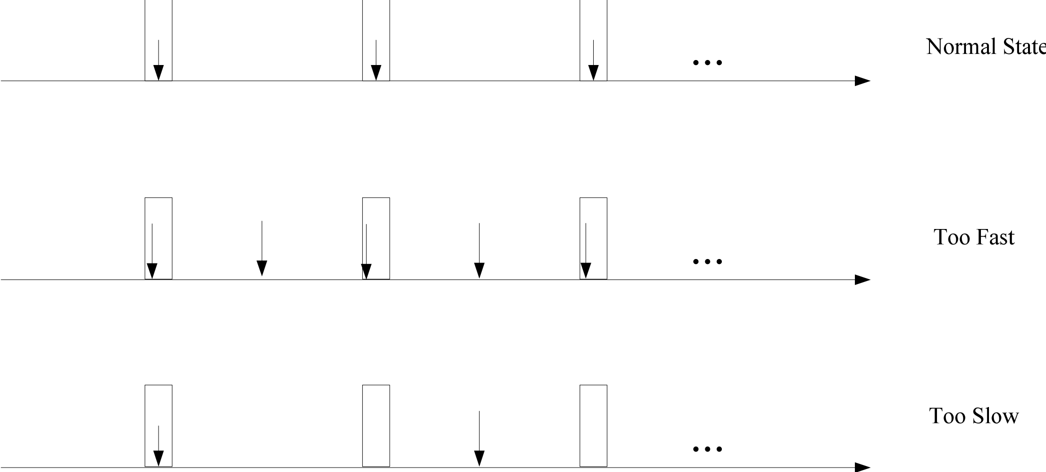

Figure 6-6 Too Fast and Too Slow Failure Modes

6.3.2 Floors and Ceilings

For any real number a, we denote the greatest integer less than or equal to "a" by $\lfloor a \rfloor$ (read "the floor of a"), and denote the smallest integer greater than or equal to "a" by $\lceil a \rceil$ (read "the ceiling of a"). For any real number "a," $a-1<\lfloor a \rfloor \leq a \leq \lceil a \rceil < a+1$.

6.3.3 Propagation Criteria

To determine if the input failure propagates to the software output, one should consider the inequality $f(I_f^t) \neq f(I_e^t)$. The inputs are expected to appear periodically with rate v_e (as shown in Figure 6-7). Then at the time $t = kT_e$, $k = 0,1,2......$, we should check if the output is the same as the expected output ($f(I_e^{kT_e})$), i.e., $f(I_f^{kT_e}) \neq f(I_e^{kT_e})$, $T_e = 1/\upsilon_e$ and $k = 0,1,2......$ The too fast/slow input rate failure may cause a too fast/slow output rate, so another propagation criterion for the rate failure is to check if the output rate is the same as the expected output rate, i.e.,

$R\left(O_f\right) \neq R\left(O_e\right)$. Then the propagation criteria for the too fast input rate can be summarized as:

$$\begin{cases} f(I_f^{kT_e}) \neq f(I_e^{kT_e}), \ \ T_e = 1/\upsilon_e \text{ and } k = 0,1,2...... \\ or \\ R\left(O_f\right) \neq R\left(O_e\right) \end{cases}$$

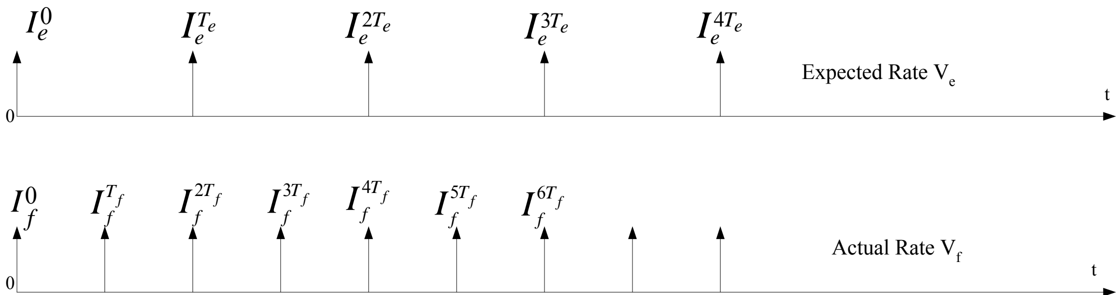

Figure 6-7 Input and Output Series

6.3.4 Too Fast Rate Failure

To study the propagation of the too fast input rate failure, we consider the following modes in which the rate failure may occur.

6.3.4.1 Sporadic Mode

In the sporadic mode, the software is triggered by an environment that is not periodic. An environmental trigger can be an interruption, an event, or other signals representing the occurrence of special condition. For instance, in a temperature monitoring system, when the temperature is higher than a critical threshold, a sensor will trigger the software's execution. In some credit card antitheft systems, when the fee of one single transaction is charged more than the limit set by the users, an alert will be sent to the user via phone or email. The signals in this case do not occur

periodically, the rate failure is not applicable to these situations. These situations are handled under the timing failure category.

6.3.4.2 Passive Mode

The set of situations occurred in the passive mode are identical to the sporadic mode except that the environment is periodic, i.e., the signals occur regularly. As shown in Figure 6-8, the interruptions arise with a period T. Once the input is ready in the environment, the software will be triggered by the interruption. The input is sent to the software along with the interruption or the software reads the input from a defined location. When the trigger rate is too fast for the software to handle all the interruptions within the time available, a too fast failure occurs.

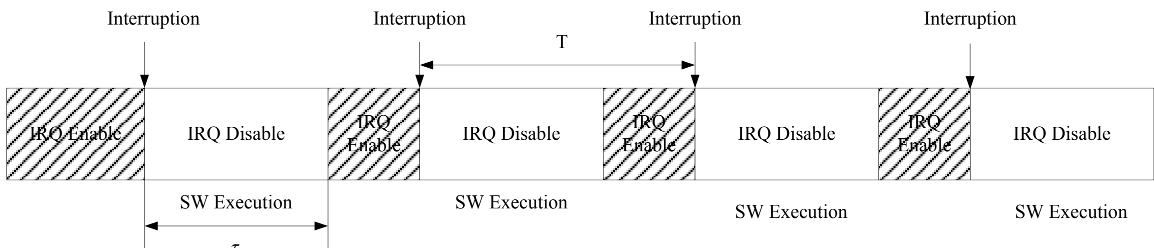

Figure 6-8 Interruption System

In the interruption system, when an interruption is being handled, the software will not be interrupted by other interruptions with equal or lower priorities. Because all the inputs have the same priority (they are the same type of input), when software is handling previous interruption, the interruption request (IRQ) is disabled. Only when the software finishes handling current interruption is the interruption request enabled.

This behavior is common in event-driven software. When an event is being handled, the new incoming events will be stored in an event task buffer. Once the software completes its current task, it picks up the next event in the buffer. When the buffer is full, no more event requests can be stored into the buffer. New incoming inputs will then be dropped.

In these systems, the software requires a minimum amount of time, denoted as τ, to execute each event request. When the actual input rate is faster than the expected maximum rate, $\upsilon_e = \frac{1}{\tau}$, a too fast input rate failure occurs. Because the event requesting rate is faster than the maximum rate, whenever the software finishes one task, it starts to handle next request. The output rate is therefore the same as its maximum output rate, i.e., $R\left(O_f\right) = \frac{1}{\tau} \neq R\left(I_e\right)$.

In the following discussion, the buffer size is denoted as n ($n \geq 0$); the actual input rate is denoted as υ_f, $\upsilon_e < \upsilon_f$.

Since the actual input rate is faster than the expected one, the buffer will be filled by the event requests sooner or later. Once the buffer is full, a new incoming event request will not be accepted. We denote the first time, at which an event request is lost, as t_l. To obtain t_l, we could assume the buffer size is n+1 instead of n. Then, the time at which an event request is stored in the n+1 position of the buffer is t_l. It can be obtained by solving the equations:

$$\begin{cases} n+1 = \lfloor \upsilon_f t_l \rfloor - \lfloor \upsilon_e t_l \rfloor \\ t_l = m_l T_f = \frac{m_l}{\upsilon_f},\ m_l = 1,2,3,...... \end{cases} \quad \textbf{(Eq. 6-2)}$$

Where

$\lfloor \upsilon_f t_l \rfloor$ is the number of total inputs arrived by time t_l,

$\lfloor \upsilon_e t_l \rfloor$ is the number of inputs which have already taken by the software by time t_l.

Hence, the first lost input occurs at time $m_l T_f$, at which the $(m_l+1)^{th}$ event request arises. The first m_l event requests are accepted by the software, in order, without missing any input. All the event requests can be handled by the software. The event request handled by software at time kτ is the actual event request stored in the buffer at time kT_f, i.e., $I_e^{kT_e} = I_f^{kT_f}$. Then the propagation criterion can be expressed as:

$$f(I_f^{kT_f}) \neq f(I_e^{k\tau}),\ k = 0,1,2...,(m_l - 1) \qquad \textbf{(Eq. 6-3)}$$

In these event driven systems, the information contained in the event requests does not change due to the too fast rate failure, i.e., $I_e^{k\tau} = I_f^{kT_f}$ and therefore $f(I_f^{kT_f}) = f(I_e^{k\tau})$. That means the too fast rate failure is therefore not detected before the m_l^{th} event request.

After $t_l - T_f$, the buffer is full. The new incoming event requests will not be accepted unless the software has taken one event request out and there is one vacancy available. Hence, the $(k+1)^{th}$ ($k > m_l$) event request with index k accepted by the software is not exactly the $(k+1)^{th}$ actual event request. Instead, it is the $(p+1)^{th}$ (p>k) actual event request with index p. If an input is stored into the buffer, it will be read by the software sooner or later. Hence, the order in which an input is stored into the buffer is the order it will be read by the software. As shown in Figure 6-9, in the period $\left[0,(m_l-1)T_f\right]$, all the inputs are stored into the buffer. Dropping inputs occurs

in the period $\left[(m_l-1)T_f, pT_f\right]$. In this period, only when an input is taken by the software, can an incoming input be accepted by the buffer. Assume p is such an input that would be stored into the buffer. In the period $\left[(m_l-1)T_f, pT_f\right]$, the number of the inputs read by the software from the buffer is $\lfloor pT_f\upsilon_e \rfloor - \lfloor (m_l-1)T_f\upsilon_e \rfloor$. That means there are $\lfloor pT_f\upsilon_e \rfloor - \lfloor (m_l-1)T_f\upsilon_e \rfloor$ inputs (including p itself) are stored into the buffer in this period. Then the corresponding index k can be expressed as:

$$k = (m_l-1) + \left(\lfloor pT_f\upsilon_e \rfloor - \lfloor (m_l-1)T_f\upsilon_e \rfloor\right),\ k \geq m_l \qquad \textbf{(Eq. 6-4)}$$

The propagation criteria after the m_l input can be expressed as:

(Eq. 6-5)

$$f\left(I_f^{pT_f}\right) \neq f\left(I_e^{kT_e}\right),\ k \geq m_l$$

where:

$$k = (m_l-1) + \left(\lfloor pT_f\upsilon_e \rfloor - \lfloor (m_l-1)T_f\upsilon_e \rfloor\right)$$

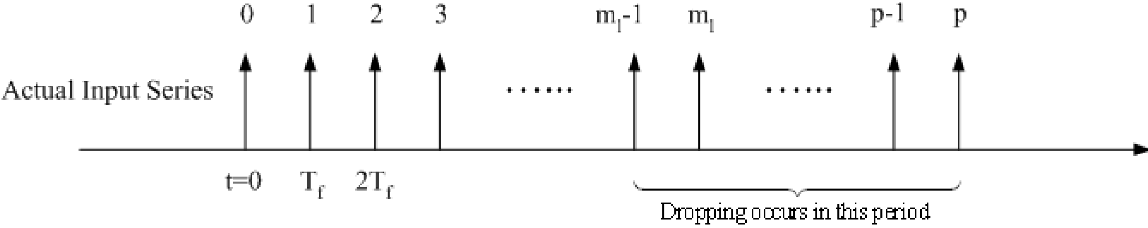

Figure 6-9 Dropped Inputs in Too Fast Rate Failure

An example is shown in Figure 6-10. In this example, the buffer size is 1; the expected input rate and the actual input rate for the event request (ER) are 3 and 2 per second, respectively.

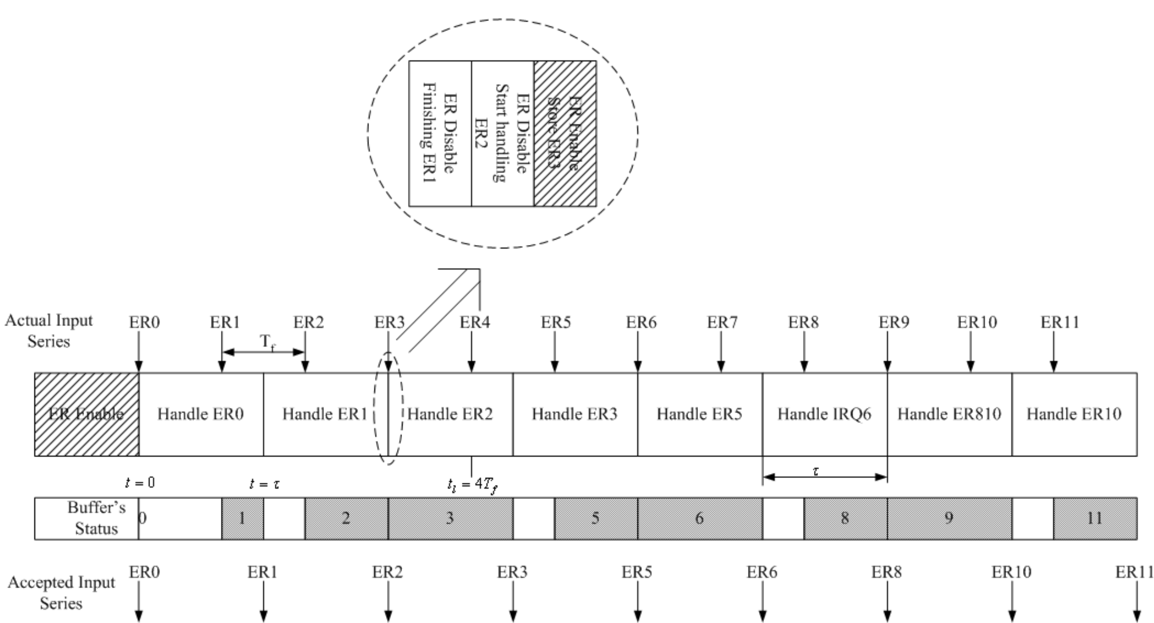

Figure 6-10 Too Fast Rate Failure in an Event Driven System (v_e=2, v_f=3, n=1)

In the above discussion, we assume that when an input (event request) arrives at the same moment at which the software just completes handling one event request and starts to handle next event request in the buffer, the event request can be stored into the buffer. For instance, at $t = 3T_f$ in Figure 6-11, the buffer is full. The software is finishing ER1 and starts handling ER2 so that there is one vacancy in the buffer and ER3 can be stored into the buffer.

The expected input rate in the above discussion is the same as the maximum physically affordable rate. In most cases, the expected input rate is slower than the maximum physically affordable rate, i.e., $\upsilon_e < \frac{1}{\tau}$. In this case, when $\frac{1}{\tau} > \upsilon_f > \upsilon_e$, all the event requests are accepted by the buffer and then further handled by the software. Whenever the software finishes handling one event request, it will generate one corresponding output. The output rate is the same as the input rate, i.e.,

$R(O_f) = R(I_f) \neq R(I_e)$. The too fast failure is thus revealed (as shown in Figure 6-11) in the output rate.

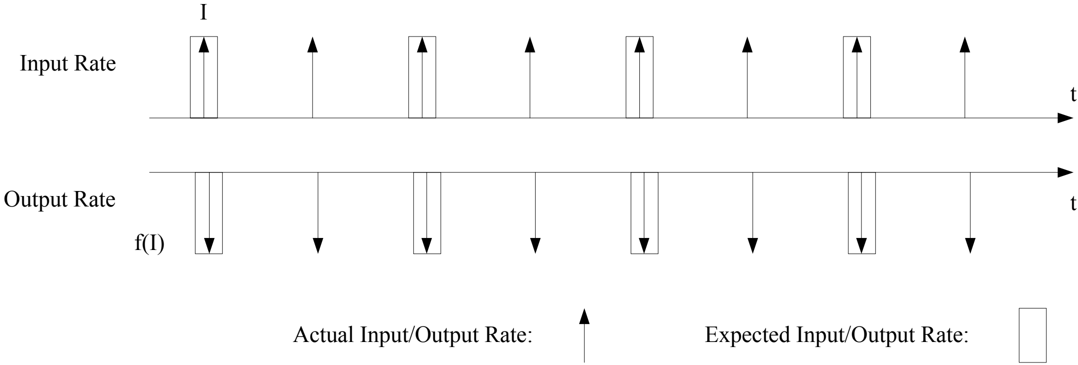

Figure 6-11 Too Fast Input Rate causes Too Fast Output Rate

6.3.4.3 Active Mode

In the active mode, the software periodically reads data from the environment. The period is controlled by a clock. In the passive mode, when the software is triggered, the input is already valid. In the active mode, when the software is triggered by a clock, an input may not be available. Because the software is triggered by its embedded clock, its output has the same rate as the clock rate. If the data in the environment is updated faster than the clock rate embedded in the software, a too fast rate failure occurs. Since the updating rate is faster than the reading rate, the input is ready every time the software reads the data. The propagation is related with the buffer size and its behavior. In the passive mode, the buffer will not accept any more inputs when it is full. However, in the active mode, the oldest input may be popped out of the buffer and the new incoming input is pushed into the buffer. In the active mode, the software is active to get the data; hence, the output rate is the same as the

software reading rate (the expected input rate) no matter the environment updating rate is fast or slow, i.e., $R(O)=R(O_e)=R(I_e)$.

There is a special active mode called the polling system. In the polling system, the software is also active to read the data in every period. However, if the data is not ready, the software keeps checking until the data is ready or the current period ends. Once the data is ready, the software handles the data and provides corresponding output and then goes to a sleeping status. In the polling system, the output rate is not so significant as the previous two cases. As long as one and only input arrives in each period, the system functions correctly. In the following sections, we will analyze these different cases.

6.3.4.3.1 Buffer Behavior 1

When the buffer is full, no more new incoming inputs are accepted. Those inputs are dropped. Only when the software reads one input from the buffer and leaves one vacancy in the buffer, can a new incoming input be put into the buffer. Figure 6-12 shows how this kind of buffer behavior impacts the fault propagation. The detailed process is described in Table 6-1.

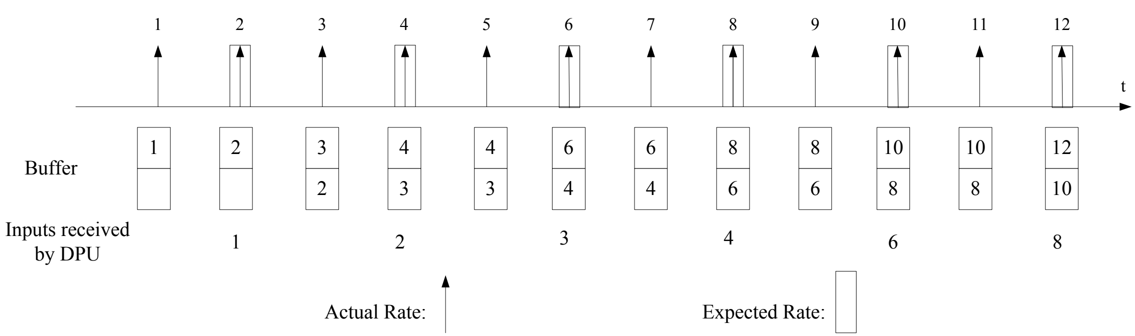

Figure 6-12 Buffer Behavior 1 (v_e=1; v_f=2; n=2)

Table 6-1 Example for Too Fast Failure (1)

Time (sec)	Behavior
0.5	The first input arrives; the time window is not activated; the first input is sent to the buffer.
1	The second input arrives; the time window is activated; the first input is read by DPU; the second input is sent to the buffer.
1.5	The third input arrives; the time window is not activated; the third input is sent to the buffer; the buffer is full.
2	The fourth input arrives; the time window is activated; the second input is read by DPU; the fourth input is sent to the buffer; the buffer is full.
2.5	The fifth input arrives; the time window is not activated; the buffer is full; the fifth input is abandoned.
3	The sixth input arrives; the time window is activated; the third input is read by DPU; the sixth input is sent to the buffer; the buffer is full.
3.5	The seventh input arrives; the time window is not activated; the buffer is full; the seventh input is abandoned.
4	The eighth input arrives; the time window is activated; the fourth input is read by DPU; the eighth input is sent to the buffer; the buffer is full.
4.5	The ninth input arrives; the time window is not activated; the buffer is full; the ninth input is abandoned.
5	The tenth input arrives; the time window is activated; the sixth input is read by DPU; the tenth input is sent to the buffer; the buffer is full.
5.5	The eleventh input arrives; the time window is not activated; the buffer is full; the eleventh input is abandoned.
6	The twelfth input arrives; the time window is activated; the eighth input is read by DPU; the twelfth input is sent to the buffer; the buffer is full.

As shown in this example, the data received by DPU at time t is not actually generated at time t. Instead, the data is generated some time before it is read. For instance, the data received by DPU at 1 second is generated at 0.5 second; the data received by DPU at 2 second is generated at 1 second. Hence, it is too early failure for each individual input data.

This case is mostly identical to the passive mode. The analysis in the passive mode is valid. However, because the data is updated by the environment, the data is time dependent. Then $I_e^{kT_e}$ may be different with $I_f^{kT_f}$. Hence, the propagation criteria can be summarized as:

$$f\left(I_f^{kT_f}\right) \neq f\left(I_e^{kT_e}\right),\ k = 0,1,2...,(m_l - 1) \quad \textbf{(Eq. 6-3)}$$

$$f\left(I_f^{pT_f}\right) \neq f\left(I_e^{kT_e}\right),\ k \geq m_l \quad \textbf{(Eq. 6-5)}$$

where:

$$k = (m_l - 1) + \left(\lfloor pT_f \upsilon_e \rfloor - \lfloor (m_l - 1) T_f \upsilon_e \rfloor\right)$$

As the example shown in Figure 6-14, ml can be calculated as 8 through (Eq. 6-2). Before the input "7", the software can handle the inputs correctly. Input "8" is the first dropped input. Input "9" can be stored into the buffer. Then, with (Eq. 6-4), one could get $k = (8-1) + \left(\left\lfloor 9 \times \frac{1}{5} \times 3 \right\rfloor - \left\lfloor (8-1) \times \frac{1}{5} \times 3 \right\rfloor\right) = 8$. That means its index in the software accepted series is 8, which is consistent with the fact.

6.3.4.3.2 Buffer Behavior 2

When the buffer is full and a new input arrives, the oldest input stored in the buffer is popped out and the new incoming input is pushed into the buffer. An example for this buffer behavior is shown in Figure 6-13. The other conditions are the same as the example presented for the buffer behavior 1. The detailed process is described in Table 6-2. One can compare it with the previous example to show how different buffer behaviors impact the inputs received by the DPU.

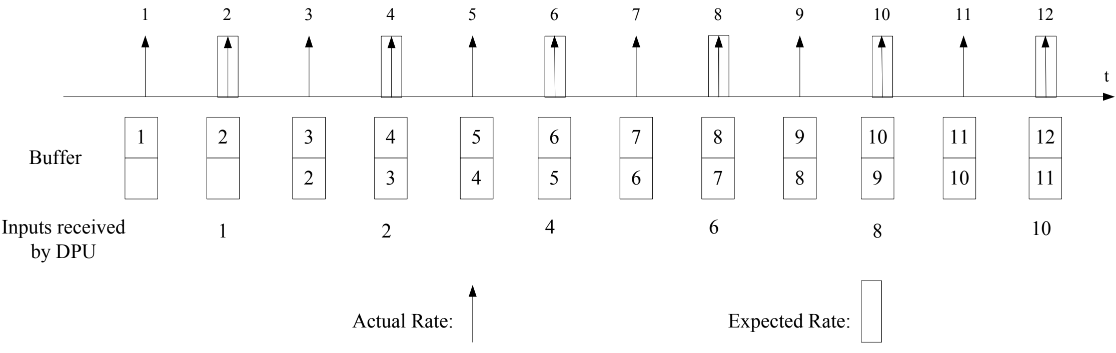

Figure 6-13 Example for Too Fast Rate Failure (2)

Table 6-2 Example for Too Fast Failure (2)

Time (sec)	Behavior
0.5	The first input arrives; the time window is not activated; the first input is sent to the buffer.
1	The second input arrives; the time window is activated; the first input is read by DPU; the second input is sent to the buffer.
1.5	The third input arrives; the time window is not activated; the third input is sent to the buffer; the buffer is full.
2	The fourth input arrives; the time window is activated; the second input is read by DPU; the fourth input is sent to the buffer; the buffer is full.
2.5	The fifth input arrives; the time window is not activated; the third input is pop out; the fifth input is sent to the buffer; the buffer is full.
3	The sixth input arrives; the time window is activated; the fourth input is read by DPU; the sixth input is sent to the buffer.
3.5	The seventh input arrives; the time window is not activated; the fifth input is pop out; the seventh input is sent to the buffer; the buffer is full.
4	The eighth input arrives; the time window is activated; the sixth input is read by DPU; the eighth input is sent to the buffer; the buffer is full.
4.5	The ninth input arrives; the time window is not activated; the seventh input is pop out; the ninth input is sent to the buffer; the buffer is full.
5	The tenth input arrives; the time window is activated; the eighth input is read by DPU; the tenth input is sent to the buffer; the buffer is full.
5.5	The eleventh input arrives; the time window is not activated; the ninth input is pop out; the eleventh input is sent to the buffer; the buffer is full.
6	The twelfth input arrives; the time window is activated; the eighth input is read by DPU; the twelfth input is sent to the buffer; the buffer is full.

Similar with buffer behavior 1, t_l is also a key parameter. Before $t_l - T_f$, all the inputs can be stored into the buffer. After $t_l - T_f$; when the new input arrives and the buffer is full, the oldest input is popped out to leave one vacancy for the new input. Before $t_l - T_f$, the software has already handled m_r inputs from the buffer which is determined by:

$$m_r = \lfloor \upsilon_e (t_l - T_f) \rfloor = \lfloor \upsilon_e (m_l - 1) T_f \rfloor \quad \textbf{(Eq 6-6)}$$

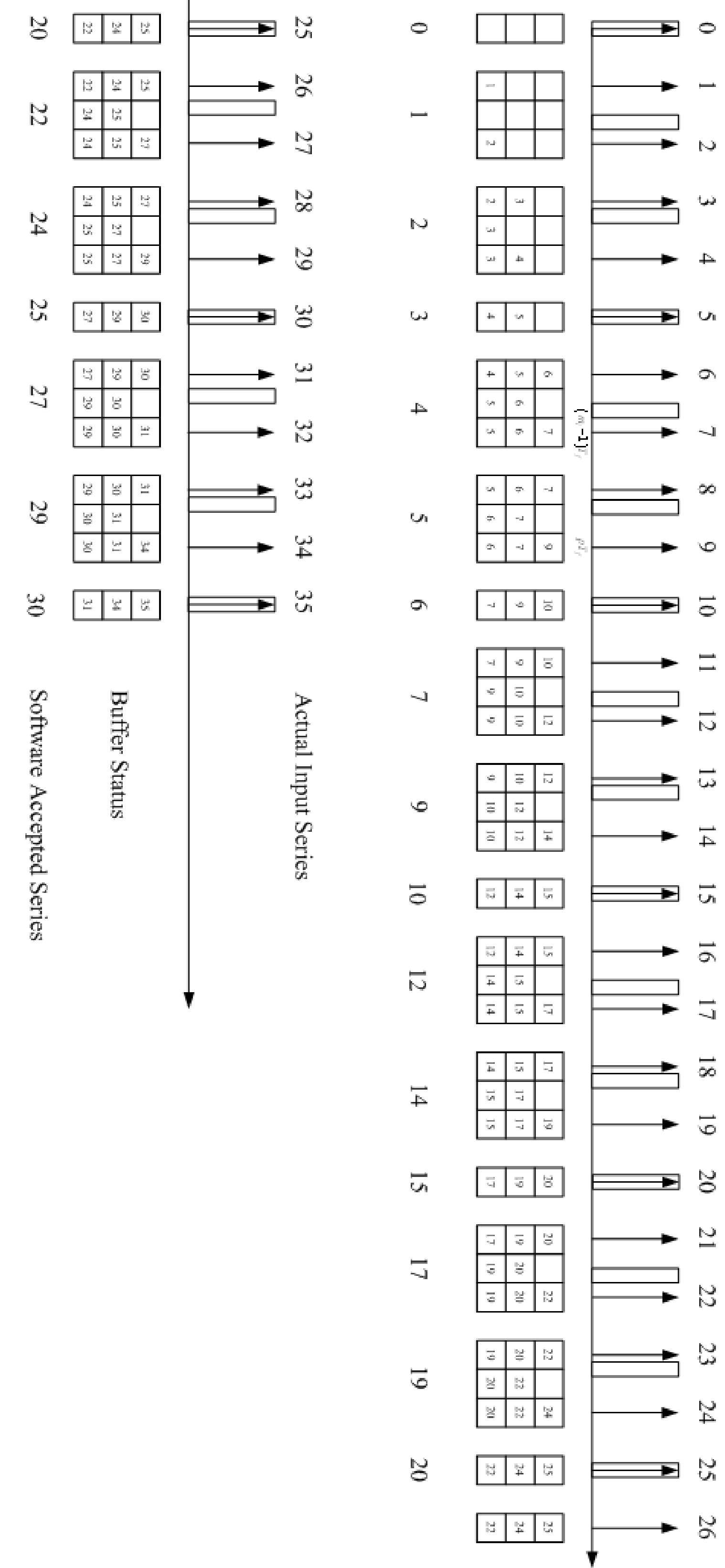

Figure 6-14 Example for Active Mode (Buffer Behavior 1)
v_e=3, v_f=5, n=3

Then the (k+1)th ($k > m_r$) input read by the software is not exactly the (k+1)th input in the actual input series. Instead, it is the (p+1)th input in the actual input series. When the software receives the (k+1)th ($k > m_r$) input from the buffer at time kT_e, some inputs have been already popped out. During the period $\left[m_r T_e, kT_e\right]$ (as shown in Figure 6-15), if the buffer is full, the oldest inputs stored in the buffer is popped out to leave one vacancy for the new incoming input. Denote the number of the popped inputs in this period as Np. Then the relationship between k and p can be expressed as:

$$(p+1) = (k+1) + N_p \Rightarrow p = k + N_p \qquad \textbf{(Eq 6-7)}$$

$$N_p = N_a^{[m_r T_e, kT_e)} - N_s^{(m_r T_e, kT_e]} \qquad \textbf{(Eq 6-8)}$$

Where

$N_s^{(m_r T_e, kT_e]} = k - m_r$ is the number of the inputs read by the software in the period $(m_r T_e, kT_e]$,

$N_a^{[m_r T_e, kT_e)}$ is the number of the inputs arrive in the period $[m_r T_e, kT_e)$.

Remember, we assume that when the buffer is read or written at the same moment, but the read action occurs ahead of the write action. Hence, $N_s^{(m_r T_e, kT_e]}$ indicates that the input read by the software at time $m_r T_e$ is not counted. Similarly, $N_a^{[m_r T_e, kT_e)}$ indicates that the input arrives at time kT_e is not counted. We introduce a positive infinitesimal ε, i.e., $\varepsilon > 0$ and $\varepsilon \to 0$, so that we could easily express $N_a^{[m_r T_e, kT_e)}$ as:

$$N_a^{[m_r T_e, kT_e)} = \left\lfloor \upsilon_f \left(kT_e - \varepsilon\right) \right\rfloor - \left\lfloor \upsilon_f \left(m_r T_e - \varepsilon\right) \right\rfloor \qquad \textbf{(Eq 6-9)}$$

Substitute $N_s^{(m_r T_e, kT_e]}$ and $N_a^{[m_r T_e, kT_e)}$ into (Eq 6-7) and (Eq 6-8), we could get:

$$p = k + N_p = k + N_a^{[m_r T_e, kT_e)} - N_s^{(m_r T_e, kT_e]}$$
$$= k + \left(\left\lfloor \upsilon_f \left(kT_e - \varepsilon \right) \right\rfloor - \left\lfloor \upsilon_f \left(m_r T_e - \varepsilon \right) \right\rfloor \right) - \left(k - m_r \right)$$
$$= m_r + \left\lfloor \upsilon_f \left(kT_e - \varepsilon \right) \right\rfloor - \left\lfloor \upsilon_f \left(m_r T_e - \varepsilon \right) \right\rfloor = m_r + \left\lfloor k \frac{\upsilon_f}{\upsilon_e} - \varepsilon \right\rfloor - \left\lfloor m_r \frac{\upsilon_f}{\upsilon_e} - \varepsilon \right\rfloor$$

Thus we have:

$$p = m_r + \left\lfloor k \frac{\upsilon_f}{\upsilon_e} - \varepsilon \right\rfloor - \left\lfloor m_r \frac{\upsilon_f}{\upsilon_e} - \varepsilon \right\rfloor \qquad \textbf{(Eq 6-10)}$$

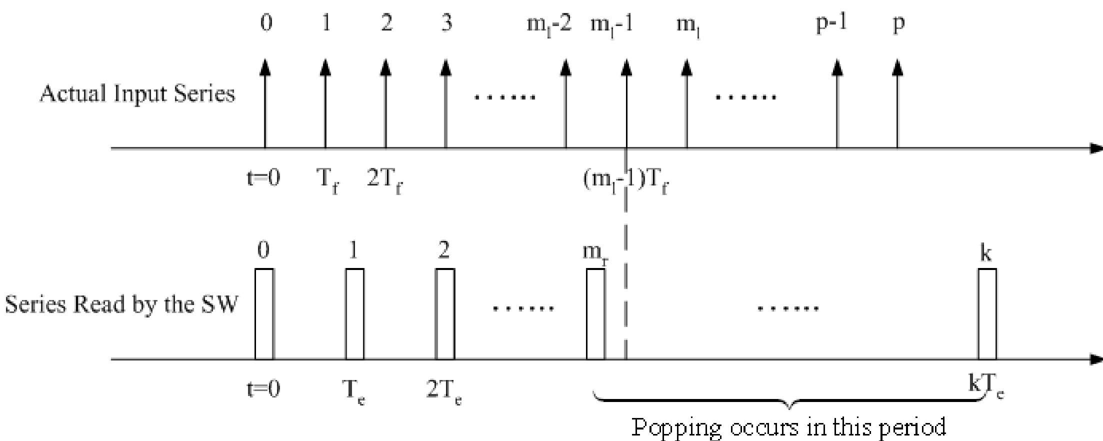

Figure 6-15 Popped Inputs in the Too Fast Rate Failure

Those equations can be proven visually in Figure 6-16. m_l can be calculated as 8 with (Eq. 6-2). Then $m_r = 4$. The input read by the software at time $8T_e$ is the input "9" in the actual input series. This is consistent with the result obtained from (Eq 6-10).

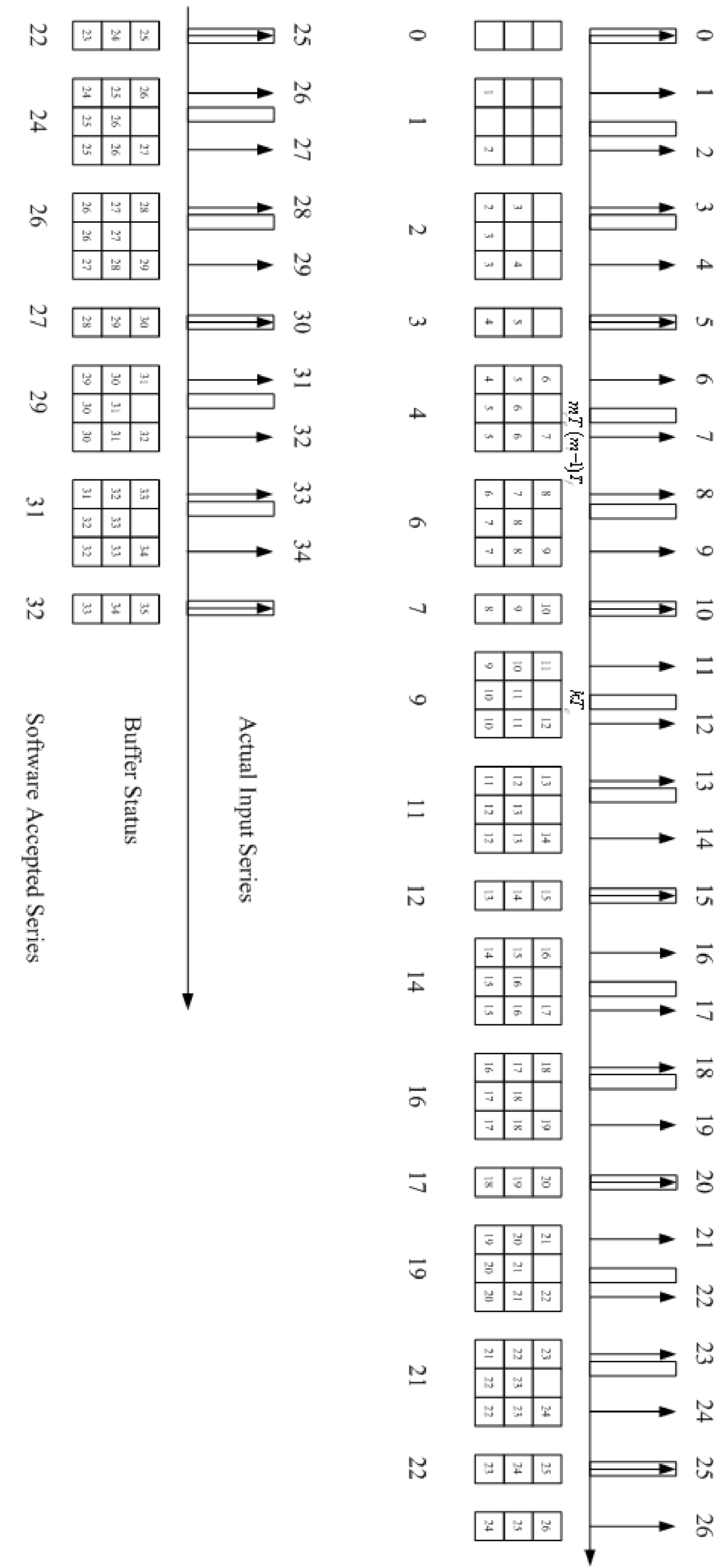

Figure 6-16 Example for Active Mode (Buffer Behavior 2)
v_e=3, v_f=5, n=3

6.3.4.3.3 Polling System

One special active mode is the polling system in which the software is set to perform tasks in one predefined period, T. At the beginning of every period, the software keeps checking if an input is valid. Once the input is valid, the software handles it and provides a corresponding output. In the remainder of the period, the software goes to a sleeping status in which it will not perform any action. The diagram for this case is shown in Figure 6-17.

The interval between two consecutive inputs should be fixed so that "the rate of input" is meaningful. Once the actual input rate, v_f, is faster than the maximum rate allowed, $\upsilon_e = \frac{1}{T}$, a too fast failure occurs.

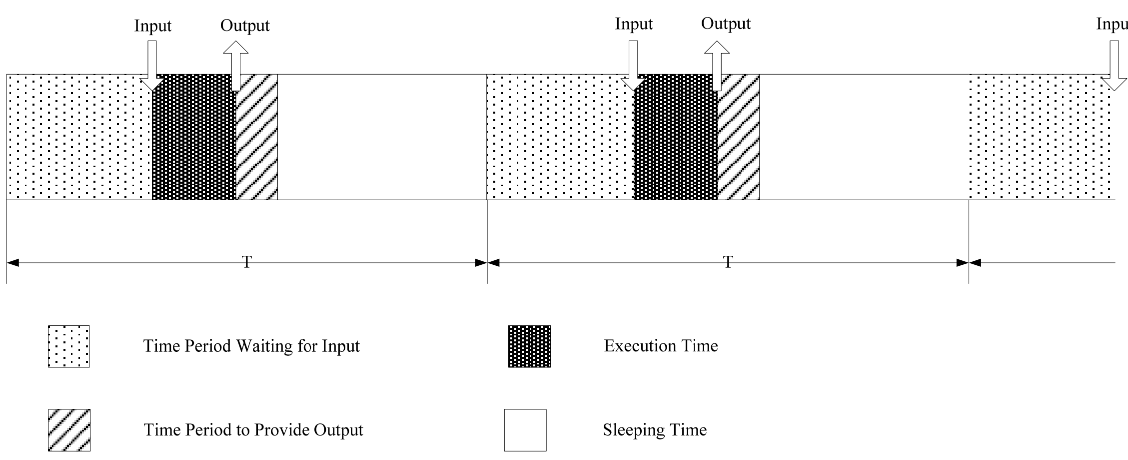

Figure 6-17 Polling System

If the execution time for one input and the time necessary to provide an output can be ignored, the polling system can be simplified as Figure 6-18.

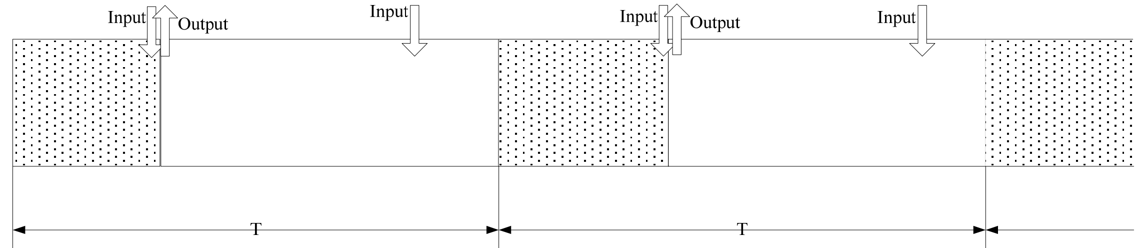

Figure 6-18 Simplified Polling System

In the simplified polling system, only one input is accepted in each period and all the other inputs are not accepted when the software is processing the input or is sleeping. This can be represented in the computational model with a buffer size always equaling 0. In each period, the accepted input is the one which arrives first in that period. In the first period, the software handles the first arrival input. In the kth (k>1) period, the software may not exactly handle the kth arrival input. In the first k period $\left[0, kT\right)$, the number of arrival inputs (not including the input arrives at kT) is $\left\lfloor (kT-\varepsilon)\upsilon_f \right\rfloor$. Here ε is a positive infinitesimal. Then, the index for of the first input accepted by the software in the (k+1)th period is $\left\lfloor (kT-\varepsilon)\upsilon_f \right\rfloor + 1$.

If $kT\upsilon_f$ is an integer, then

$$\left\lfloor (kT-\varepsilon)\upsilon_f \right\rfloor + 1 = \left\lfloor kT\upsilon_f - \varepsilon \right\rfloor + 1 = kT\upsilon_f - 1 + 1 = kT\upsilon_f = \left\lceil kT\upsilon_f \right\rceil$$

If $kT\upsilon_f$ is not an integer, then

$$\begin{aligned} &\left\lfloor (kT-\varepsilon)\upsilon_f \right\rfloor + 1 = \left\lfloor kT\upsilon_f - \varepsilon \right\rfloor + 1 = \left\lfloor \left\lfloor kT\upsilon_f \right\rfloor + \varepsilon \right\rfloor + 1 \\ &= \left\lfloor kT\upsilon_f \right\rfloor + 1 = \left\lceil kT\upsilon_f \right\rceil \end{aligned}$$

Hence, in the (k+1)th period, the index of the first input accepted by the software is $\left\lceil k\upsilon_f T \right\rceil$.

If the input rate is the expected input rate, then in the (k+1)th period, the index of the input should be (k+1). Hence, the propagation criterion can be expressed as:

$$f(I_f^{\lceil k\upsilon_f T\rceil T_f}) \neq f(I_e^{(k+1)T_e}),\ k=1,2,3...... \quad \textbf{(Eq 6-11)}$$

When the software execution time and the time to provide output can not be ignored, the analysis is more complex. The end of a period is the deadline for completely providing the output. The deadline to receive the input is $T-\tau_e-\tau_p$, where τ_e and τ_p are the software execution time and the time to provide the output respectively. The too fast failure may cause the software to receive the first input after the deadline in some periods. Hence, the propagation criteria could be expanded to:

$$\begin{aligned} & f(I_f^{\lceil (k-1)\upsilon_f T\rceil T_f}) \neq f(I_e^{kT_e}) \\ & or \\ & (kT-\lceil (k-1)\upsilon_f T\rceil T_f) < T-\tau_e-\tau_p \\ & k=1,2,3...... \end{aligned} \quad \textbf{(Eq 6-12)}$$

6.3.5 Too Slow Failure Mode

When the actual rate is slower than the expected rate, a too slow failure occurs (as shown in Figure 6-19). From a microcosmical viewpoint, every single input is later than its expected arrival time.

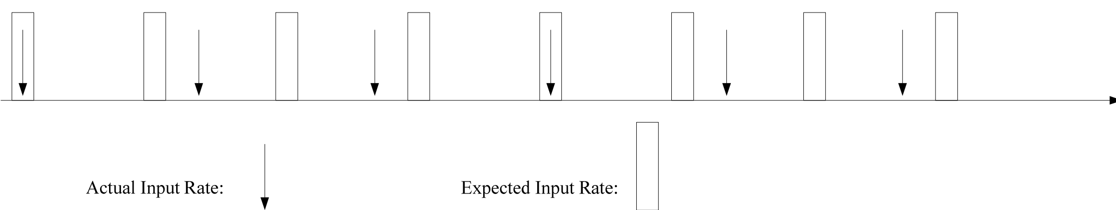

Figure 6-19 Too Slow Failure

6.3.5.1 Passive Mode

Because the rate is slower than the expected rate, all the inputs in the "too slow failure mode" will be handled by the system. However, the system will generate an output series with a slow rate. Macrocosmically, the output is stretched along the time dimension (as shown in Figure 6-20). At time $t = kT_e,\ k = 0,1,2,......$, the input should be I^k. However, with the too slow input rate, at time $t = kT_e,\ k = 1,2,......$, the input to the software may be empty or pT_f if pT_f equals to kT_e, i.e.,

$$I_f^{kT_e} = \begin{cases} I^p,\ pT_f = kT_e \\ \varnothing,\ \text{otherwise} \end{cases}$$

In the passive mode, when and only when an input arrives, the software can perform a corresponding action. Then the output rate is the same as the too slow input rate, i.e., $R(O_f) = R(I_f) \neq R(I_e)$. Therefore, the propagation probability for this case is 1.

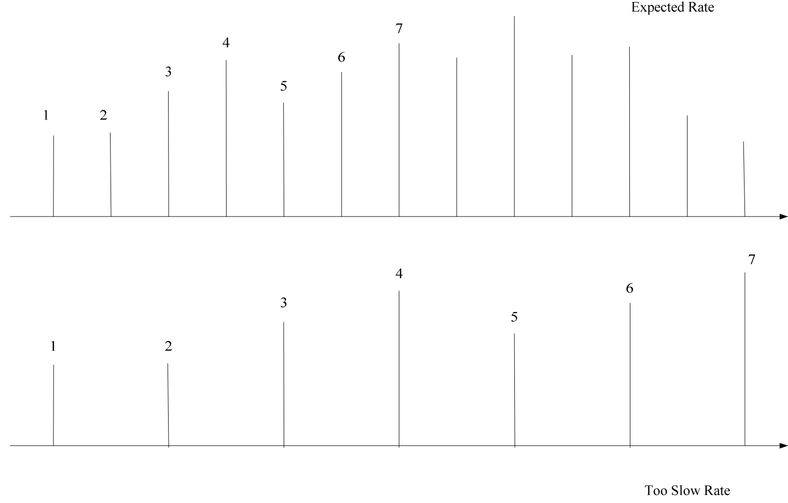

Figure 6-20 Time Stretch

6.3.5.2 Active Mode

Since the actual updating rate is slower than the expected rate, whenever the software reads data from the environment, the data may not be ready or it is outdated and does not reflect current environment status. For instance, in a temperature monitoring system, when the sensor's updating rate is slower than the expected rate, the software will not be able to read data that it is as new as possible (as shown in Figure 6-21). At time t, where $t = kT_e = k/\upsilon_e, k = 0,1,2......$, the data has been updated $\lfloor t\upsilon_f \rfloor$ times which occurred at time $\frac{\lfloor t\upsilon_f \rfloor}{\upsilon_f}$. Hence, the criterion failure is propagation to the software output can be expressed as:

$$f(I_f^{\lfloor t\upsilon_f \rfloor / \upsilon_f}) \neq f(I_e^t), t = kT_e = k/\upsilon_e, k = 0,1,2...... \qquad \textbf{(Eq 6-13)}$$

Result from the expected rate — Result from the too slow rate — Field Data

Figure 6-21 Too Slow Failure in the Temperature Monitoring System

In some cases, if the data is not ready when the software tries to read it, the software may use a predefined value for processing. As we discussed in the timing failure, the predefined value could be a constant or a dynamic value. Actually, in the above temperature monitoring system, the software does not care if the data is ready. It keeps reading a predefined value which is dynamically updated by the environment.

If the software uses a constant when the data is not ready, then the propagation criteria can be expressed as:

$$\begin{cases} f\left(I_e^{kT_e}\right) \neq f\left(I_f^{pT_f}\right), pT_f = kT_e, p, k \in N^+ \\ f\left(I_e^{kT_e}\right) \neq f\left(c\right), \text{ otherwise} \end{cases} \quad \textbf{(Eq 6-14)}$$

No matter the software uses constant or dynamic value, it always has data to process. Hence, the output rate is the same as the software reading rate which is the expected output rate, i.e., $R\left(O_f\right) = R\left(O_e\right)$.

6.3.5.3 Polling System

When the input rate is slower than 1/T, a too slow failure occurs. If in one period, there is no input due to the too slow input rate, a failure will be detected. Let us assume the input "k" is the first input from which the too slow input rate failure starts to be detected (as shown in Figure 6-22). Then between this input and its previous input "k-1", there must be an entire period $\left[pT_e, \left(p+1\right)T_e\right]$. Then we have:

$$\left(k-1\right)T_f < pT_e$$
$$kT_f \geq \left(p+1\right)T_e$$

Where $k \in N^+$, $p \in N$

Because "k" is the first input from which the too slow input rate failure starts to be detected, the first k inputs (0, 1, 2, ……, k-1) distribute in the first k periods. One input falls in one period. That implies: p=k. Replace p with k in the above inequalities, one could get:

$$\frac{T_e}{T_f - T_e} \leq k < \frac{T_f}{T_f - T_e}, \; k \in N^+ \quad \textbf{(Eq 6-15)}$$

Thus, before (k+1)th input, the too slow input rate failure can not be detected in the polling system. When the input rate is slower than 1/T, it is impossible that two consecutive inputs arrive in the same period. Then without considering the deadline for the inputs, the software will provide the output in the same rate as the input rate, i.e., $R(O_f) = R(I_f)$.

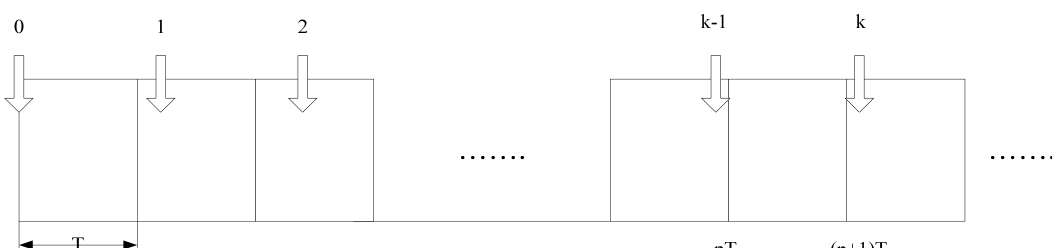

Figure 6-22 Too Slow Failure in Polling System

6.3.6 Summary

The propagation behaviors of the too fast and too slow rate failure are summarized in Table 6-4 and Table 6-3 respectively.

Table 6-3 Summary for the propagation of too slow failure

Mode		Description of Propagation Behavior
Passive Mode		The too slow input rate causes a too slow output rate. The output rate is the same as the input rate. The propagation probability is 1.
Active Mode	Dynamical value	The propagation depends on (Eq 6-13). The output rate is the same as the software reading rate.
	Constant	The propagation depends on (Eq 6-14). The output rate is the same as the software reading rate.
	Polling System	The too slow input rate causes a too slow output rate. The propagation probability is 1.

Table 6-4 Summary for the propagation of too fast rate failure

Mode		Description of Propagation Behavior
Passive Mode	The input rate is faster than the maximum physical rate, $\frac{1}{\tau}$, which the software can handle.	The too fast input rate failure is not detected before the first m_l inputs are handled. After that, the propagation depends on (Eq. 6-5). The output rate is $\frac{1}{\tau}$.
	The input rate is slower than $\frac{1}{\tau}$, but faster than the expected rate.	The too fast input rate causes a too fast output rate. The output rate is the same as the input rate. The propagation probability is 1.
Active Mode	The new incoming inputs will be dropped if the buffer is full.	For the first m_l inputs, the propagation depends on the (Eq. 6-3), for the remaining inputs, the propagation depends on the (Eq. 6-5). The output rate is the same as the expected rate.
	If the buffer is full, the oldest input in the buffer is popped out and the new incoming input is pushed into the buffer.	For the first m_r inputs, the propagation depends on the (Eq. 6-3), for the remaining inputs, the propagation depends on the (Eq 6-10). The output rate is the same as the expected rate.
	Polling System with period T	The too fast input rate failure is not detected in the first period. From the second period, the propagation depends on the (Eq 6-12).

6.4 *Duration Failure*

6.4.1 *Definition*

The duration is the time period during which the input lasts. In [2, 3], duration is formally defined as:

$$D(I) = \text{the amount of time during which the input, I, is defined}.$$

6.4.2 *Too short failure & too long failure*

When the duration of an input is shorter than the lower bound of the expected input duration, τ, ($D(I) < \tau$), a "too short duration failure" occurs (as shown in Figure 6-23a). When the duration of an input is longer than the upper bound of the

expected input duration, σ, ($D(I) > \sigma$), a "too long duration" failure occurs (as shown in Figure 6-23b). The initiator of a duration failure may be a hardware component, software component, or a human being.

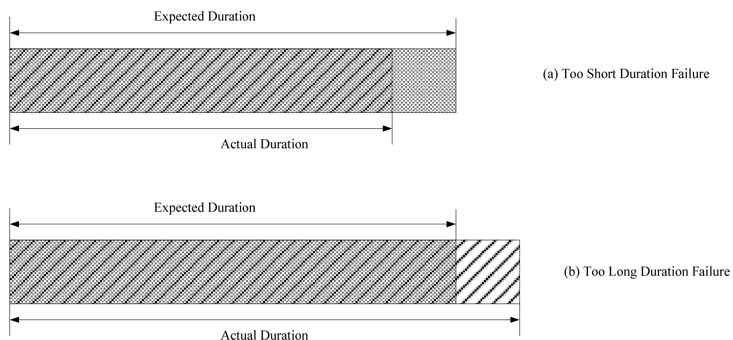

Figure 6-23 Duration Failure

6.4.3 Role of the Duration

The role of the duration characteristic in the software environment should be studied before analyzing how a duration failure propagates. Normally, the duration of an input may be used in two modes:

1. The measure of the duration is used as the value of a variable defined later in the software. For instance, the exposure duration is one of the parameters used in calculating radiation dose. When the duration is used as one of the input parameters, it must be measured by the software. According to the definition of data state in section 4.1, the too short /long duration failure causes an incorrect data state error directly.

2. The duration acts as a means of identifying the presence of an input. For instance, in an interrupt system, the trigger duration must be longer than a certain limit so that the interruption can be identified. In most cases, the

duration is used as the means to recognize the existence of an input. The too short and too long failure will affect directly the input identification.

In the following sections, we discuss the propagation of the duration failure in these two modes.

6.4.3.1 Too Short Duration Failure

If the duration is used in mode 1, let us denote by val_d the variable related with the measurement of the duration. Then the "too short duration" causes a smaller measurement value, i.e., $val_{d,f} < val_{d,e}$, where $val_{d,f}$ and $val_{d,e}$ represent the values of the too short duration and the expected duration. Hence the propagation of the "too short duration" depends on the propagation of the incorrect value, val_d :

$$f\left(val_{d,f}\right) \neq f\left(val_{d,e}\right),\ val_{d,f} < val_{d,e} \qquad \textbf{(Eq. 6-16)}$$

If the duration is used in mode 2, the input with too short duration will not be identified by the software. For example, when providing an input from the keyboard, if the keystroke action is too quick (the duration of the input is too short), the input will not be detected. Another example is the impulse counter. To avoid miscounting false signals, a threshold is often set for the pulse duration. If the duration of an input is shorter than the threshold of the pulse duration τ, i.e., $D\left(I\right) < \tau$, a too short duration failure occurs and the input is not counted. Hence, an input with too short pulse duration will be missed. Since the input is not detected, it seems that an omission failure occurs or in other words, the too short duration failure is transformed into an omission failure.

Based on the above analysis, one can summarize the propagation of the too short duration failure as shown in Figure 6-24.

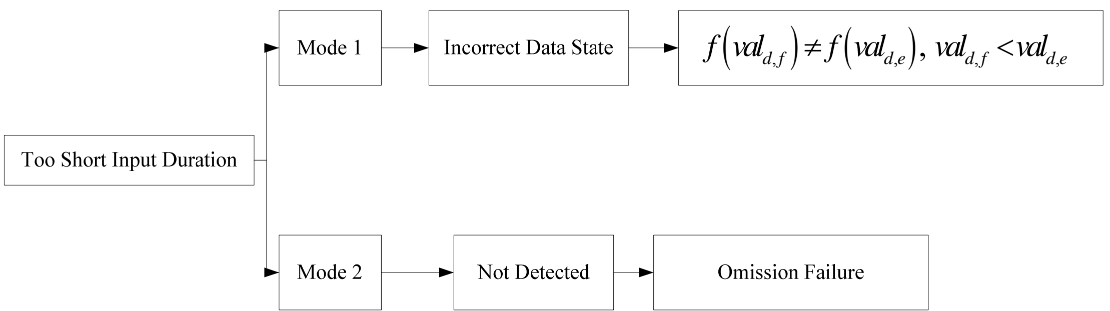

Figure 6-24 Propagation of the "Too Short" Duration Failure

6.4.3.2 Too Long Duration Failure

If the duration is used in mode 1, the "too long duration failure" causes a larger measurement value, i.e., $val_{d,f} > val_{d,e}$. The larger value may exceed the range of the expected value. If the value is not out of range, the propagation can be expressed as:

$$f\left(val_{d,f}\right) \neq f\left(val_{d,e}\right),\ val_{d,f} > val_{d,e} \qquad \textbf{(Eq. 6-17)}$$

If the value is out of range, because the value is non-address-related, the value may or may not be adjusted as described in 5.2.2.

If the duration is used in mode 2, the too long duration may lead to the occurrence of multiple inputs (as shown in Figure 6-25). For instance, if key "a" is pressed too long, the software may regard the input as several inputs "a". The number of redundant inputs is related to the system configuration. Generally, let us denote by d_r the duration used to identify an input. Then the number of redundant inputs is $N_r = \frac{D(I)}{d_r} - 1$. The propagation of the too long duration failure depends on how the software treats these redundant inputs.

In practice, the software developers may use some "special marks" to avoid responding to redundant inputs. For instance, when the key "a" is pressed too long, the operating system generates a series of "key down" events and only one "key up" event when the key is released. In the implementation, the software may be designed to respond only to the event "key up" instead of to the event "key down" to identify that a key is clicked. If no such "special marks" are used, all the redundant inputs are sent to the software along with the expected input. Those redundant inputs are stored in the buffer (size N). The number of redundant inputs may be larger than the buffer size, so the number of redundant inputs which can be handled is $\min\left(N, N_r\right)$. As shown in Figure 6-26, those redundant inputs may cause input failures in the next inputs.

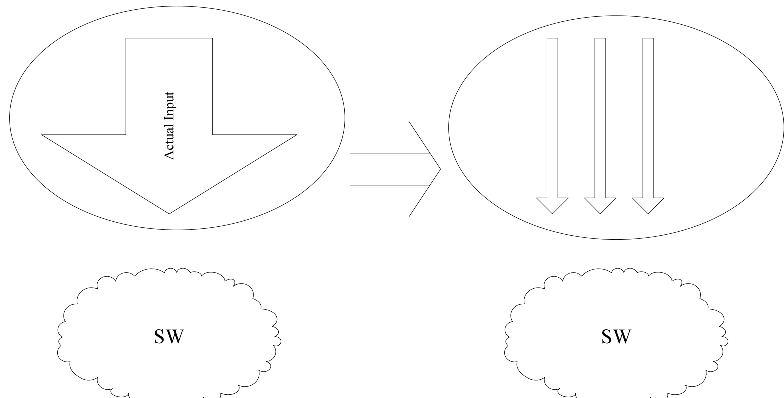

Figure 6-25 Input with “Too Long” Duration is Treated as Several Inputs

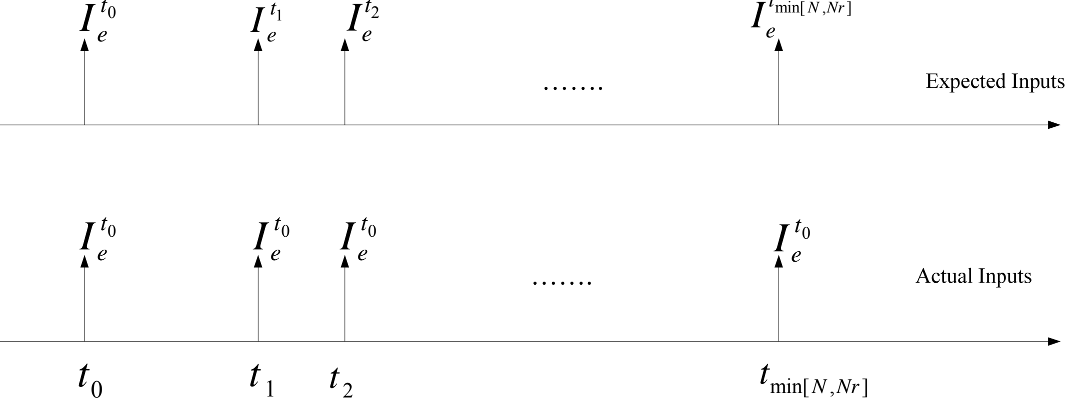

Figure 6-26 Propagation Illustration of Redundant Inputs

The first redundant input will be used by the software at time t_1 at which the first input after the current expected input is expected to arise. At t_1, the expected input should be $I_e^{t_1}$, so the propagation criterion for the first redundant input is $f\left(I_e^{t_1}\right) \neq f\left(I_e^{t_0}\right)$. Similarly, the propagation criterion for the second redundant input is $f\left(I_e^{t_2}\right) \neq f\left(I_e^{t_0}\right)$, so forth and so on. Then the propagation criterion for the redundant inputs can be expressed as:

$$f\left(I_e^{t_i}\right) \neq f\left(I_e^{t_0}\right),\ i = 1, 2, \ldots\ldots, \min\left(N, N_r\right) \qquad \textbf{(Eq. 6-18)}$$

Where

t_i is the time at which the ith expected input after the current expected input is expected to arise.

From the above discussion, one can see that if duration is considered as a variable, the too long duration failure will cause a value failure. The value failure may be out of the range of the expected value. The incorrect value and out of range value will cause incorrect data state. If the duration is used as the means to identify the input, the too long duration failure will generate a series of redundant inputs along with the expected input. If the software is designed to respond only to the "special

mark", the input with too long duration will be identified as one input (expected input). Otherwise, the redundant inputs are sent to the software and may cause incorrect outputs. Those propagation behaviors are summarized in Figure 6-27.

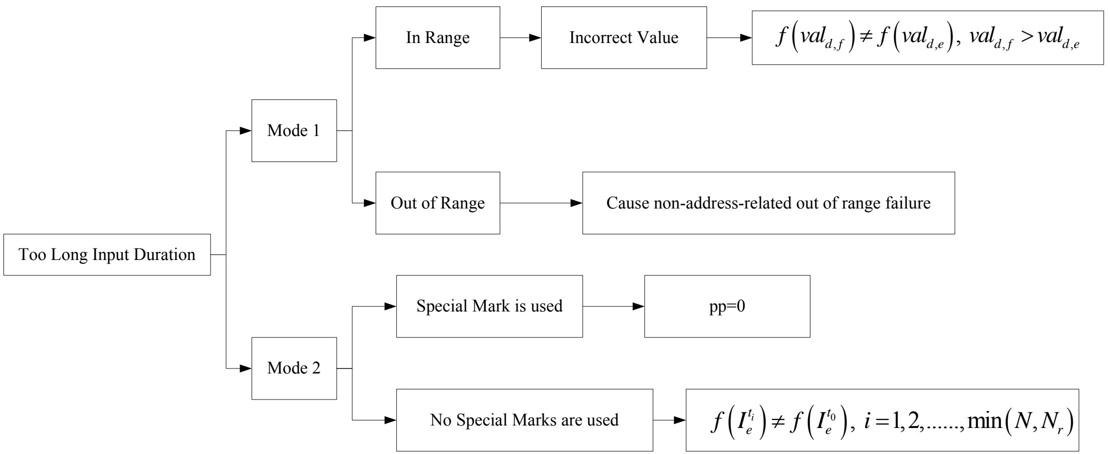

Figure 6-27 Propagation of the “Too Long” Duration Failure

7 Application

In this chapter, we discuss how the fault propagation analysis can be applied to quantify the contribution of input failures to the risk by using a personal access control software (PACS) as example.

7.1 PACS

PACS is a simplified version of an automated Personal entry/exit Access System (PACS) used to provide privileged physical access to rooms /buildings, etc. The functioning of PACS is summarized as follows: A user inserts his personal ID card that contains his name and social security number into a reader. The system searches for a match in the software system database which may be periodically updated by a system administrator, instructs/disallows the user to enter his personal identification number, a 4 digit code using a display attached to a simple 12 position keyboard, validates/invalidates the code, and finally instructs/disallows entry into/exit out of the room/building through a gate. A single line display screen provides instructional messages to the user. An attending security officer monitors a duplicate message on his console with override capability [36, 37].

The PACS software can be divided into four components: "SwipeCard", "EnterPIN", "Process", and "SeeOfficer". The component "SwipeCard" is used to validate provided card information. We applied the fault propagation analysis only on "SwipeCard". The functional requirements for "SwipeCard" are described below:

i. *The software waits for some user to swipe his/her ID card in the Card Reader, by waiting on register R6 to be set to 1 by the Card Reader – denoting that some ID card has been swept through the Card Reader; clears the register R6 by writing 0;*

ii. *Then, waits for register R10 to be set to 1, denoting that the information from the card has been successfully read by the hardware. The software will wait a maximum of 5 seconds after R6 was set to 1 for R10 to become active (be set to 1). In case of timeout the software will assume that the card was unreadable, and the message "RETRY" will be sent to the user.*

iii. *If R10 is set to 1 within 5 seconds indicating that the card is readable, the software requires a minimum of 1 second to be ready to receive the card information sent from the card reader.*

iv. *In case of successful reading of the card, the software will check the security file "card.val" to determine whether the user is allowed to gain physical access. If the social security number and the last name don't match with the entry in this file, or if the entry doesn't exist the software will assume that there is some error with the card, and the message "RETRY" will be sent to the user. If three attempts have been tried, the message "See Officer" will be sent to the user and the card reader will be locked and wait for a reset action.*

7.2 *Application process*

This section discusses how to apply propagation analysis to help quantify the contribution of the input failures to risk.

In addition, this section also shows that the systematic application of propagation analysis for the different input failure modes reduces the number of test cases necessary for the evaluation of the propagation probability. Actually, if the propagation behavior of a failure mode has been confirmed through fault propagation analysis, i.e., its propagation probability is 0 or 1, it becomes unnecessary to evaluate the impact of this failure mode through test. The steps required to quantify the contribution of the input failures to risk are shown in Figure 7-1.

Step 1. Construct f the Artifact.

Step 2. Identify the expected inputs for the scenario.

Step 3. Identify the applicable input failure modes.

Step 4. Identify the input failure modes whose analytical propagation probabilities are 1 or 0.

Step 5. Determine the operational profile.

Step 6. Model the failure modes whose propagation behaviors can not be determined by propagation analysis using TestMaster; generate test cases with profile coverage.

Step 7. Execute the test cases in WinRunner and calculate the software failure probability due to the input failures.

Figure 7-1 Application Process

Taking "SwipeCard" as an example, we illustrate the details of each step:

Step 1: Construct the artifact.

The available materials for this example are: the system requirement (SyRS), the software requirements specification (SRS), a fire hazard report, and a manufacturing manual for CardReader. The SyRS and SRS provide information on how to use the PACS software to exit the building. The fire hazard report gives additional information such as the fact that the computer system will normally stop working 10 minutes into a fire accident. The manufacturing manual provides information on the performance of the CardReader under special conditions.

Step 2: Identify the expected inputs for the scenario.

The event sequence diagram (ESD) for the PACS exit system is shown in Figure 7-2. The scenario under study is: a fire event has occurred; the fire protection system did not work; the emergency exit was closed; the guard was not in position behind the desk. The scenario is marked in the Figure 7-2.

If the software component “SwipeCard” fails and the guard is absent, the PACS software can not be reset to let another user out. The exit is thus blocked and this will lead to the end state “Loss of Occupants”.

The computer will stop working 10 minutes into the fire accident. However, as the SRS indicates, the software allows a maximum of 5 seconds to read the card information successfully. Otherwise, it will prompt an error message and wait for a reset action of the guard. Because the guard is not in position behind the desk, the reset cannot be performed and the PACS software is thus stuck and a failure “Loss of

Occupants" occurs. Hence, as soon as a card information becomes erroneous, the system will experience a failure.

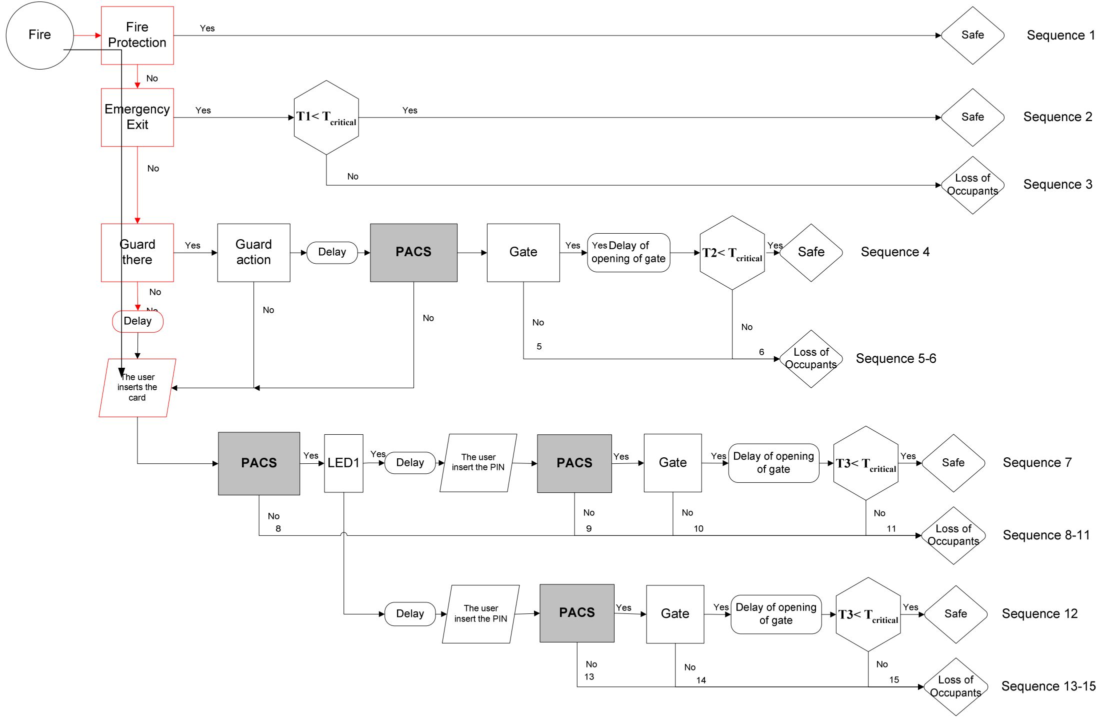

Figure 7-2 ESD for Exit System

Then, the expected inputs for the software component "SwipeCard" are the correct cards before 10 minutes – time necessary to process the entrant. In the SRS, the characteristics of the expected inputs for "SwipeCard" are specified as shown in Table 7-1.

Table 7-1 Characteristics of SwipeCard

Characteristics	**Specification**
Value	The information stored in the security file
Type	Characters array for both SSN and LastName
Amount	2 (SSN and LastName)
Range	Exact 9 characters for SSN, up to 20 characters for LastName
Timing	Maximum 5 seconds for reading the card information successfully. Minimum 1 second for reading the card information if the card is readable.
Duration	Not specified.
Rate	Not specified.

Step 3: Identify the applicable input failure modes.

Not all the input failure modes are applicable for any given software component. The applicability of the input failure modes to "SwipeCard" is described in Table 7-2.

Table 7-2 Applicability of the Failure Modes

Input Characteristic	Potential Failure Mode	Description
Value	Incorrect Value	The card information consists of SSN and LastName. Only when both SSN and LastName match with the record in the security file, is the card considered as a correct card. When the information is not listed in the record, a value failure occurs.
Type	Type Mismatch	Both SSN and LastName are character arrays. The input will be treated as character by the function used in the software. So the type failure is not applicable to PACS.
Range	Out of Range	SSN and LastName are not numeric type but type *char* and range failure is applicable only when the input is of numeric type, so the range failure is not applicable to PACS.
Amount	Too little, Too much	The card information is comprised of the SSN and LastName. If either of these two parameters is missing, a “too little” amount failure occurs. If other information is provided with the SSN and LastName, a “too much” amount failure occurs.
Rate	Too fast, Too slow	The inputs to the component do not occur periodically. So the rate failure is not applicable to PACS.
Duration	Too long, Too short	If the card is swiped too quickly, the CardReader can not read the information encoded on the card, and a too short duration failure occurs. When the card is swiped slowly, the card reader can still read the card information correctly, so the “too long” duration failure is not applicable.
Time	Too early, Too late, omitted	If the card information can not be read successfully within 5 seconds, a timeout occurs. If the card reader sends the card information in less than 1 second after R10 is set to 1, a too early failure occurs.

Step 4: Identify the input failure modes whose analytical propagation probabilities are 1 or 0.

According to the fault propagation analysis, some failure modes will propagate to the output in some situations, while some modes will not. Let us denote by $\mathscr{I}_{f,1}$ and $\mathscr{I}_{f,0}$ the failure modes whose propagation probabilities can be determined by fault propagation analysis as 1 and 0 respectively. The remaining failure modes whose propagation behaviors are not determinable by fault propagation analysis are denoted by $\mathscr{I}_{f,0.5}$. Apparently, the set of input failures is the union of $\mathscr{I}_{f,0}$, $\mathscr{I}_{f,0.5}$, $\mathscr{I}_{f,1}$.

Value Failure:

As the fault propagation analysis carried out in previous chapterss indicates, the propagation probability of a value failure is not determinable if no other information is provided.

Timing Failure:

If the card information can not be read successfully within 5 seconds, a "too late" failure occurs. Figure 7-3 describes the propagation of the "too late" failure. In this example, an error detection mechanism is used (case 2) and it will send message "Timeout" and "Retry" to the user. This message is different from the expected message "EnterPIN". The "too late" failure is thus propagated.

If the card reader sends the card information to the software less than 1 second after R10 is set to 1, a "too early" failure occurs. As shown in Figure 7-4, because there is no buffer designed for the premature input, the input will not be accepted by the software. Then, it will cause a "too late" failure. Hence, both the "too late" and the "too early" failures are propagated to the output as shown in Figure 7-5.

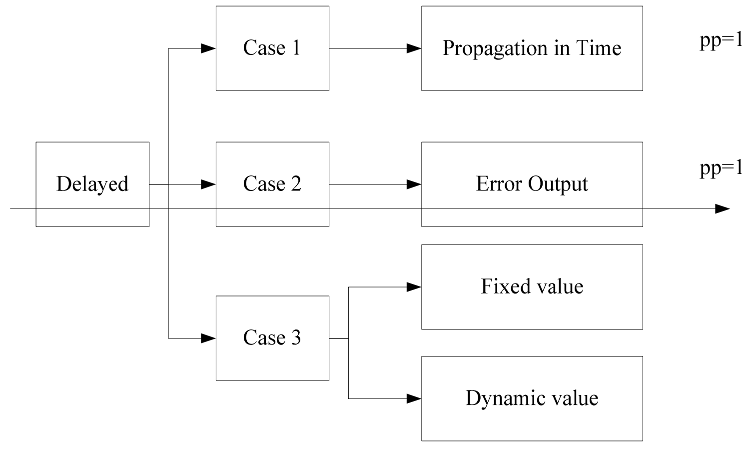

Figure 7-3 Propagation of "too late" failure

Too Early
Case 1
Case 2
Case 3
With buffer
Without buffer
Input will be used when time window is activated
pp=0
Input will be accepted
Cause a Too Late Failure
Error Output
pp=1
Depends on Recovery Mechanism

Figure 7-4 Propagation of "too early" failure

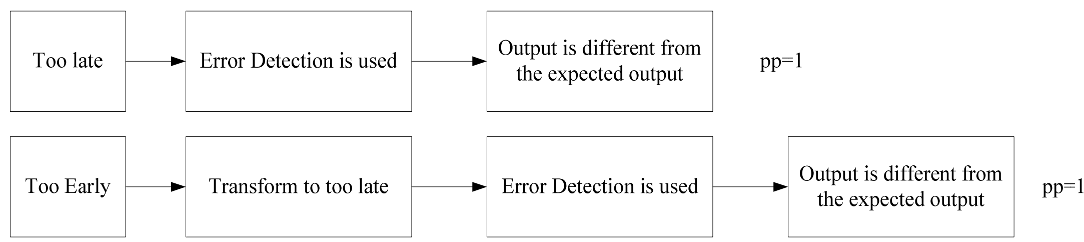

Figure 7-5 Propagation of timing failures in SwipeCard

Duration Failure:

The original propagation analysis for the “too short” duration failure is shown in Figure 7-6. In this example, the duration is used in mode 2, i.e., used to identify the existence of an input. Then, the “too short” duration failure will be transformed into the “omitted” failure which will propagate to the software output.

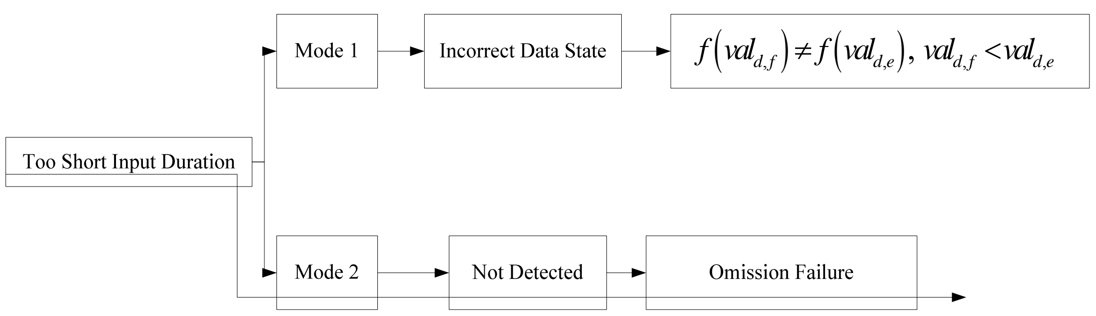

Figure 7-6 Propagation of the "too short" duration failure

Amount Failure:

The two input elements SSN and LastName are read from the same component "CardReader" and SSN is read before LastName. As shown in Figure 7-7, SSN and LastName come from the same component, missing either of them will cause a "too little" amount failure.

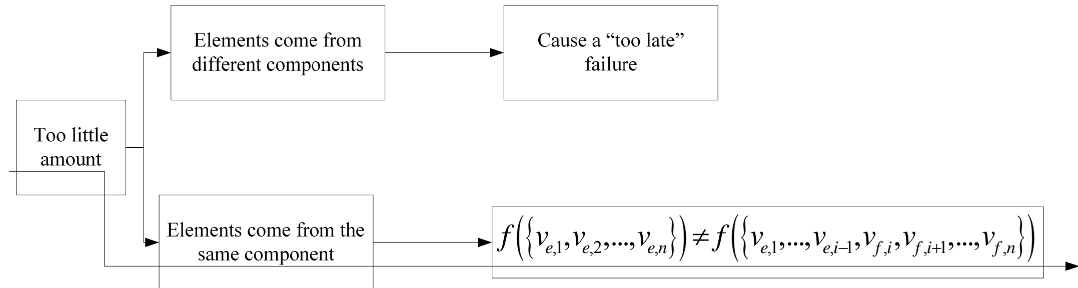

Figure 7-7 Propagation of the "too little" Failure

If both SSN and LastName are missing, then the "too little" amount failure is transformed to the "omitted" failure which will propagate to the output. If only SSN is missing, part or all of the LastName will be read as SSN. Because SSN contains only numeric characters while LastName contains alphabetical characters, the input can not pass the validation stage and the failure propagates to the output. If LastName is missing, the input can not pass the validation stage either because there is no record with only SSN in the security file. Hence, the "too little" amount failure will propagate to the output (as shown in Figure 7-8).

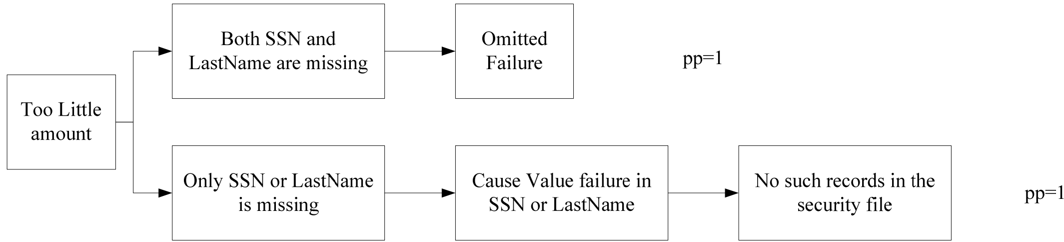

Figure 7-8 Propagation of the “too little” amount failure in SwipeCard

If additional elements are provided to the software, a “too much” amount failure occurs. The general propagation behavior of the “too much” amount failure is shown in Figure 7-9. In this example, all the inputs (SSN and LastName) come from the same device (CardReader); they are not read in the fixed format (the cardreader may read up to 20 characters for the LastName). If the additional elements are inserted after SSN, they will be treated as part of LastName and cause a value failure to LastName. On the other hand, if the additional element is inserted before SSN, it will cause a value failure to SSN and LastName (as shown in Figure 7-10). Since the propagation for the value failure is not determinable, the propagation for the “too much” amount failure can not be determined only using fault propagation analysis.

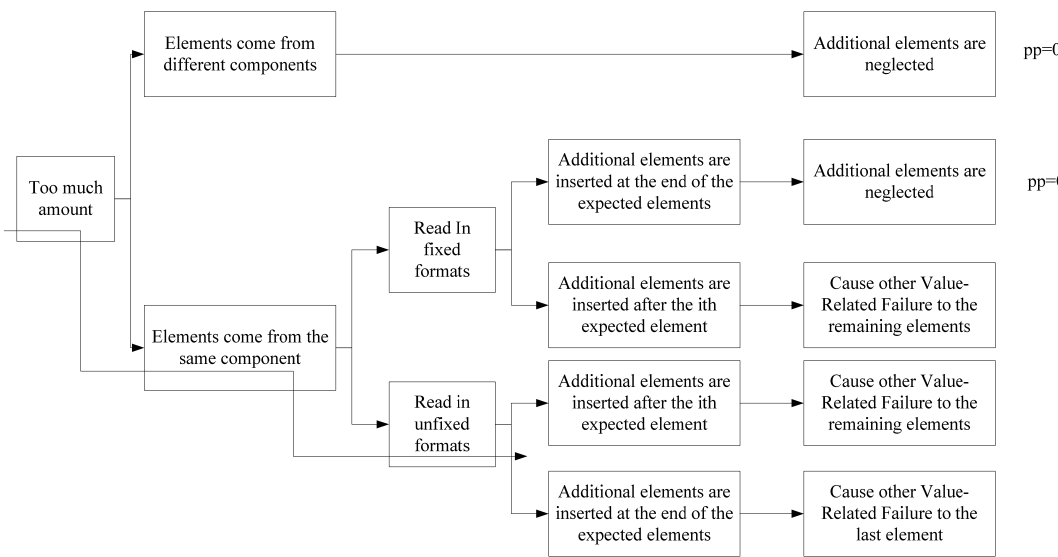

Figure 7-9 General propagation behavior of the "too much" amount failure

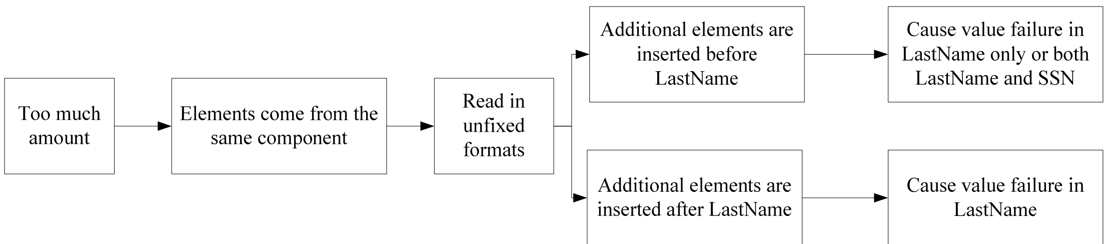

Figure 7-10 Propagation of the "too much" amount failure in SwipeCard

The result of fault propagation analysis for the applicable failure modes on "SwipeCard" is summarized in Table 7-3.

Table 7-3 Result of Fault Propagation Analysis on "SwipeCard"

Failure Mode	Fault Propagation Analysis Result
Incorrect value	Not Determinable
Too little amount	Propagates
Too much amount	Not Determinable
Too late	Propagates
Too early	Propagates
Too short duration	Propagates

Step 5: Determine the operational profile

The upstream component is the CardReader. According to the manufacturing manual, the CardReader performs correctly with a probability 99.5%. Incorrect performance will cause incorrect values in 80% of the cases, missing part or all information in 4% of the cases, providing additional information in 3% of the cases, providing data too late in 10% of the cases, providing data too early in 1% of the cases or omitting data in 2% of the cases. The operational profile of the input failures is shown in Table 7-4.

Table 7-4 Operation Profile for SwipeCard

Failure Mode		Probability
Value	Incorrect Value	0.8
	Too little amount	0.04
	Too much amount	0.03
Time	Too late	0.10

	Too early	0.01
Duration	Too Short	0.02

Step 6: Model the input failures with their associated operational profile

As the analysis in step 3 indicates, we only need to model the "too much" amount failure and the "incorrect value" failure, because their propagation cannot be determined only using fault propagation analysis. The failure modes, $FM \in \mathcal{I}_{f,0.5}$, and their associated operational profiles are modeled using TestMaster [38].[4] Because it is unnecessary to model the failure modes in the set $\mathcal{I}_{f,1}$ and $\mathcal{I}_{f,0}$ whose propagation behavior is readily known through fault propagation analysis, the number of test cases is reduced. The TestMaster models are shown in Figure 7-11 to Figure 7-14.

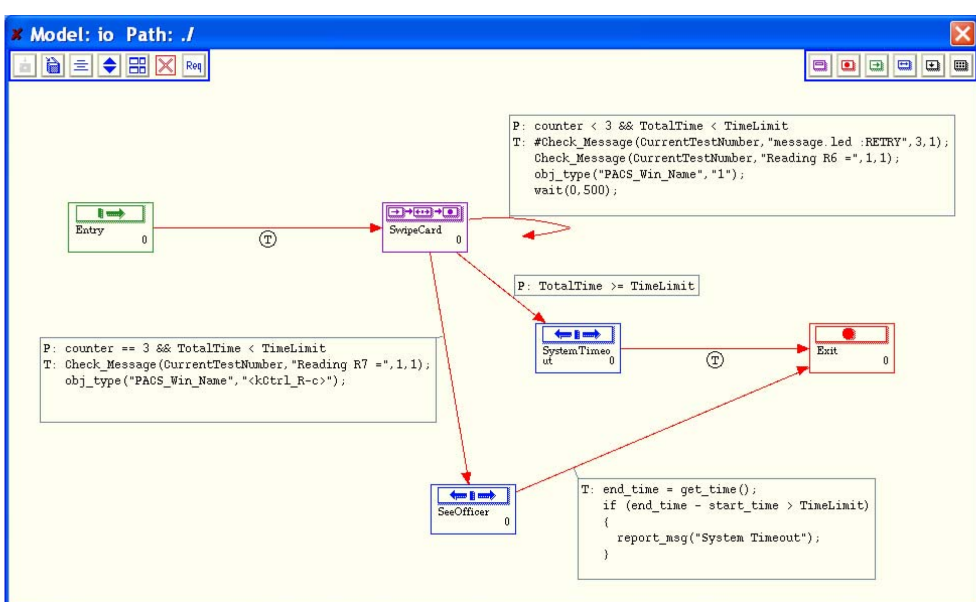

Figure 7-11 TestMaster Model - Mainframe

[4] Please refer to the Appendices for a short introduction to system modeling with TestMaster.

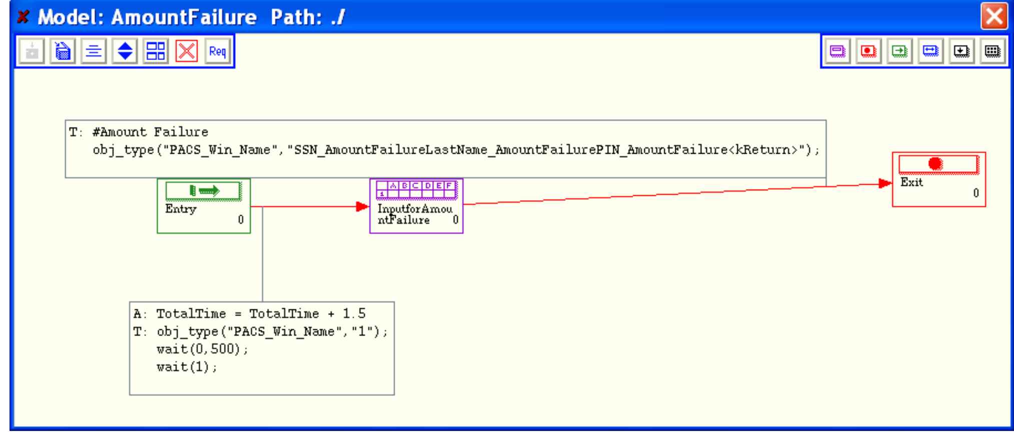

Figure 7-12 TestMaster - Amount Failure

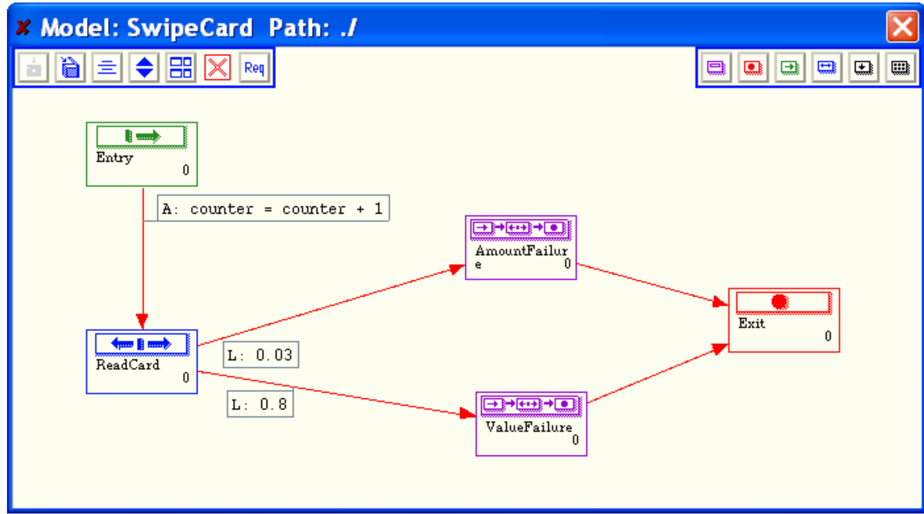

Figure 7-13 TestMaster Model – SwipeCard

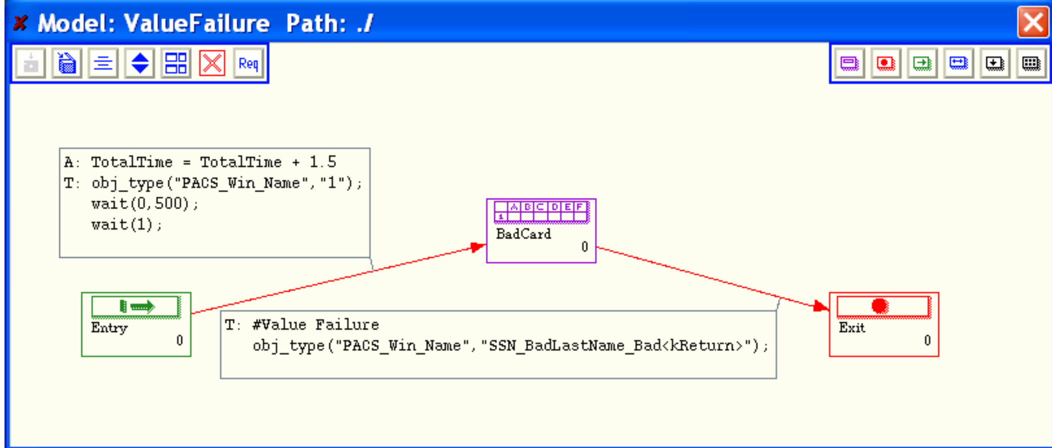

Figure 7-14 TestMaster Model - Value Failure

Step 7: Execute test cases in WinRunner [39][5] and calculate the software failure probability due to input failures. The test cases generated from TestMaster are automatically executed in WinRunner if they are written in the WinRunner TSL script.

After running the test cases, the probability that the software component fails given that an input failure has occurred can be calculated with:

$$prob = p\left(I_f \in \mathcal{I}_{f,1}\right) + p\left(I_f \in \mathcal{I}_{f,0.5}\right)\frac{N_f}{N}$$

Where

N is the total number of test cases generated,

N_f is the number of failures observed,

$p\left(I_f \in \mathcal{I}_{f,1}\right)$ is the probability that an input failure comes from set $\mathcal{I}_{f,1}$,

$p\left(I_f \in \mathcal{I}_{f,0.5}\right)$ is the probability that an input failure comes from set $\mathcal{I}_{f,0.5}$.

In this example, there are two PMAs which will map to "Enter PIN" and "See Officer" respectively. The expected input will fall into the PMA corresponding to "Enter PIN", so an input failure propagates to the output if its output is "See Officer". Hence, in this example, the propagation probability equals to the probability that the input failure generates an output "See Officer".

Two hundred test cases are generated and used to test the application in this example, of which 195 fail (i.e., generate output "See Officer"). Hence, the probability that the software component "SwipeCard" will fail given the occurrence of an input failure is:

---

[5] Please refer to the Appendices for a description of an approach that can be used to automate testing with WinRunner.

$$prob = p\left(I_f \in \mathscr{I}_{f,1}\right) + p\left(I_f \in \mathscr{I}_{f,0.5}\right)\frac{N_f}{N} = 0.17 + 0.83 \times \frac{195}{200} = 0.989$$

The confidence interval can be also estimated with:

$$\frac{p + \frac{1}{2N} z_{1-\alpha/2}^2 \pm z_{1-\alpha/2}\sqrt{\frac{p(1-p)}{N} + \frac{z_{1-\alpha/2}^2}{4N^2}}}{1 + \frac{1}{N} z_{1-\alpha/2}^2}$$

Where α is the confidence level, N is the sample size, p is the estimation of the probability. In this example, p is 0.989 and N is $200/0.83 = 241$. The uncertainty comes from the 200 test cases, because the portion of the input failure whose propagation probability is 1 does not contain any uncertainty. Then, the 95% confidence interval can be calculated as [0.952, 0.993].

From this example, one can see that it is not necessary to model the "too little" amount failure, the "too short" duration failure, the "too early" failure, and the "too late" failure. Hence, no test cases are generated and run for those failure modes. Because they correspond to 17% of all the input failures in this example, 41 (200/0.83-200) test cases are saved.

8 Conclusion and Future Research

In this chapter, we conclude on the benefits that fault propagation analysis can bring to probabilistic risk assessment (PRA). We also identify the shortcomings and limitations of methods presented in this dissertation. Avenues for future research are subsequently identified.

8.1 *Advantages of fault propagation analysis*

The methods presented in this dissertation have the following benefits::

1. They provide an approach for the identification of the input failures of a software component.
2. The Image Reconstruction Method provides a way to quantify the contribution of input failures to risk. Compared with random testing, the method can help reduce the number of test cases necessary for the evaluation of the propagation probability. The fault propagation analysis shows that some types of input failures will definitely propagate to the software output if a set of given conditions are met. It is unnecessary to generate test cases for such inputs.
3. The Image Reconstruction Method could provide useful information for the propagation analysis of the failures of the support platform on which the software resides. The support failures may cause a data state error in an arbitrary location of the software. As described in section 4.5.5, with the Image Reconstruction Method, it is possible to identify the flat parts in

the locations where the support failure arises by simply mapping the flat parts in the input domain to such locations. Hence, it is not necessary to repeat the application of the Image Reconstruction Method in those locations.

8.2 *Limitation of the Methods and Future Research*

The methods presented in this dissertation have the following limitations:

1. The analysis assumes that the software is deterministic, i.e., the software provides identical output given identical inputs. The analysis is invalid if the software is not deterministic.
2. Although the propagation analysis has accounted for as many architectures (i.e. with buffer, without buffer) as possible while examining the behavior of timing and rate failures, it is impossible to guarantee the completeness of the conditions studied. However, the analysis process is valid for uncovered cases. The key point is to analyze how the input failures cause data state errors so that the propagation criteria can be formalized.
3. Identifying the flat parts is the key process of the Image Reconstruction Method. Improving the identification mechanism will have a significant impact on the calculation efficiency. Such study will be the focus of future research.
4. Currently, the fault propagation analysis is limited to considering single input failures. However, multiple failures are possible. For instance, in the "too long" duration failure, when a "special mark" is used to avoid

redundant inputs, a possible delay may ensue. The analysis for multiple input failures is much more complex than that for the single input failure. The propagation criteria may not be a simple union of the propagation criteria for each single failure. The possible solution could be 1) to design a procedure to clearly identify possible multiple failures, 2) to study the interaction among multiple failures and identify new impacts other than the impact from a single failure, and then 3) to formalize the propagation criteria for multiple failures.

Appendice A

User Guide for TestMaster and WinRunner

This section takes PACS as an example to discuss the modeling of the software system/components. For more information about TestMaster, please refer to the TestMaster's User Guide [1].

TestMaster is a test design tool that uses the extended finite state machine notation to model a system. To model the software components and generate test cases using TestMaster, 3 steps need to be performed.

1. Create Models.
2. Define Transitions.
3. Generate Test Cases.

1. Create Models

First, we need to start TestMaster from the UNIX command line by typing:

tm [option] [projectName]

In our example, we typed: tm pacs.tp. Then the TestMaster windows are shown as displayed in Figure A-1.

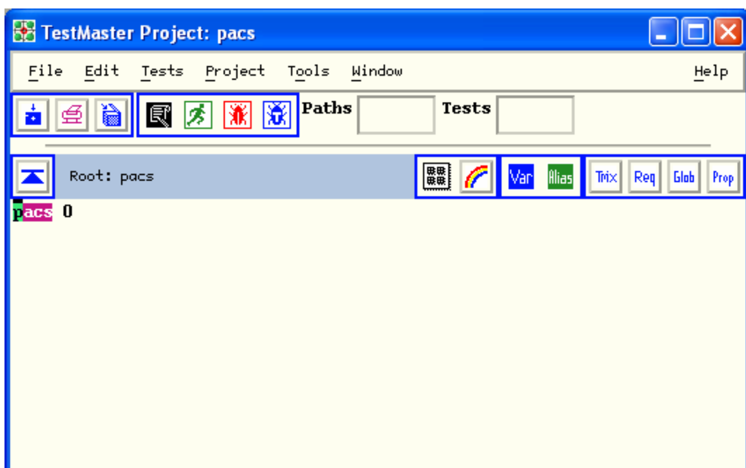

Figure A-1 TestMaster Project Window

TestMaster is started with a Root Model as shown in Figure A-2.

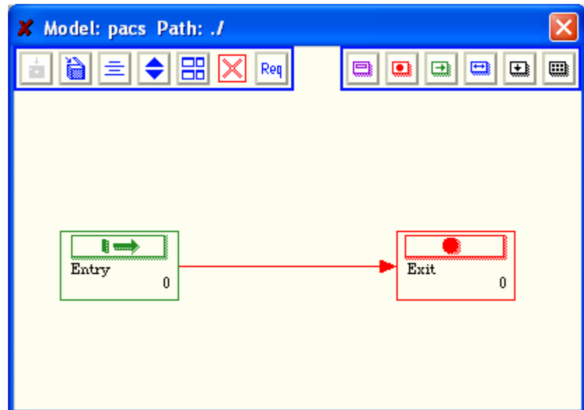

Figure A-2 Entry Level (Root) Model

According to the SRS of PACS, we divided the PACS software into WaitsForCardInReader, SwipeCard, EnterPIN, Proceed, SeeOfficer, and BeforeExit sub-models. We added these sub-models by creating state and model calls as shown in Figure A-3. WaitsForCardInReader is used to model the R6 hardware failure before swiping the card. It should be noted that TestMaster allows the construction of hierarchical finite state machines. States can not be decomposed further whereas model calls can be further decomposed.

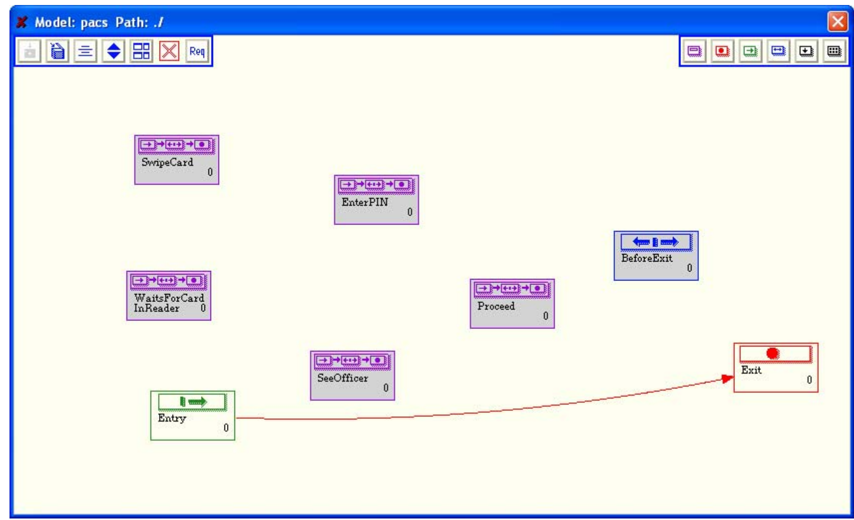

Figure A-3 Create Objects in TestMaster

Besides the model call and state, there are other modeling objects in the toolbar located at the top right hand section of the model window. Each object represents a different function in TestMaster, but each object is created, edited, and deleted exactly the same way. See Table A-1 for details on each object.

Table A-1 Description of Object Toolbar

Object	Icon	Description
State		A state represents a point in time, or a mode of existence, and is displayed on the Model window as a box.
Model Call		A state which is a call or reference to another Model. This is analogous to a function call in a programming language.
Table Model		A Table Model is used to enter values for variables. Variables created for a Table Model in the Project window, are automatically positioned in the top row of the Table. Each row in the Table Model performs exactly like a Transition.
Entry		The first state in every Model. No paths are generated without an entry state.
Exit		The last state in a Model or submodel. When the Test Generator reaches this State, control returns to the calling Model, if any.
C_LIMB		Used to model a C-based Application Programming Interface using C-Language Interface Model Builder (C_LIMB). C_LIMB automatically generates C_language code for each routine of the API being tested.

2. Define Transitions

Objects created in TestMaster are isolated of each other unless they are connected with at least one transition.

A transition represents the events required to drive the test from a particular State or Model Call to the next State or Model Call. To create a transition, perform the following steps:

1. Position the mouse in the center of a State, Model Call or Table Model icon that is the starting point of the Transition, and click the middle mouse button.
2. Move the cursor to the connecting State or Model Call.
3. Click the middle mouse button in the center. The Transition is created.

Transition information can be found either explicitly or implicitly in the SRS document. For example, the following statements from PACS SRS (see Appendix A) indicate transitions between sub-models as shown in Table A-2.

Table A-2 SRS Statements Indication

Statement in the SRS	**Tansitions Between Sub-Models**	
	Origin	Destination
The user should swipe the ID Card in the Card Reader.	WaitsForCardInReader	SwipeCard
PIN error, the user should see the officer.	EnterPIN	SeeOfficer
ID Card error, the user should see the officer.	SwipeCard	SeeOfficer
Upon successful reading of the ID card and finding a match in the security file, the user is asked to enter his PIN.	SwiperCard	EnterPIN
The ID Card and PIN match, the user is allowed access.	EnterPIN	Proceed

After adding the transitions, the highest level PACS model obtained is given in Figure A-4.

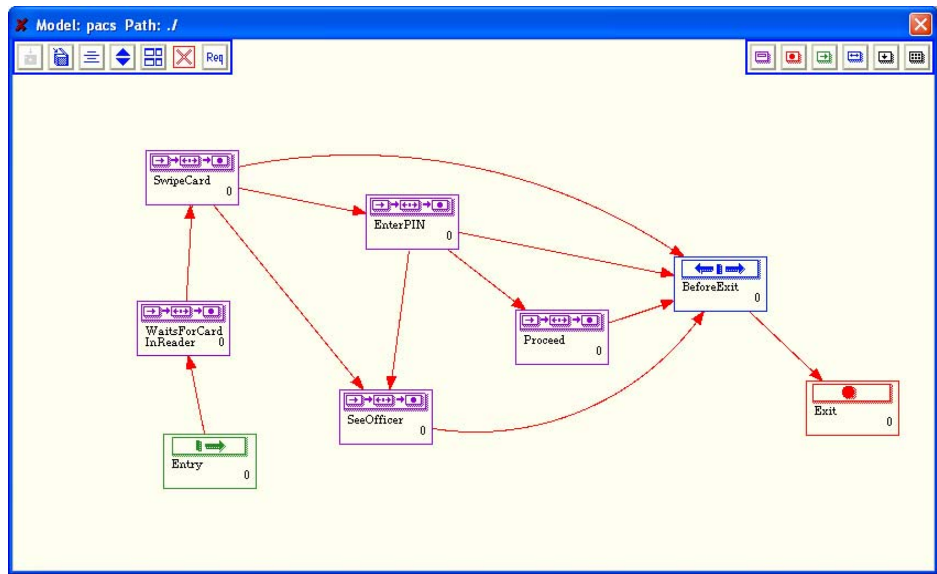

Figure A-4 Adding Transitions

More transitions may be added in the future if needed. The next step is to define the transitions in detail. Position the mouse on the transition you want to edit and right click the mouse. Choose edit transition from the drop menu, the edit transition window pops up as shown in Figure A-5.

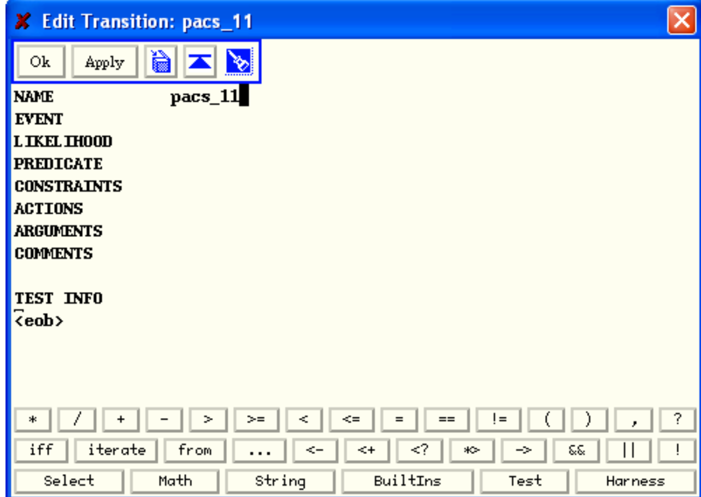

Figure A-5 Edit Transition Window

A transition contains several fields including name, event, likelihood, predicate, constraints, actions, arguments, comment, and TEST INFO. The major fields are described in Table A-3.

Table A-3 Major Fields Description in a Transition

Field	Description
Likelihood	Likelihood is a field used to define the Relative Likelihood of Occurrence (RLOC) for a Transition. In the Likelihood field, a PFL Expression is entered; whose result when evaluated is an absolute number between 1 and infinity.
Predicates	Predicates are specifications placed on transitions as a PFL expression to specify the behavior of the current transition and move the test generator to the next state or model call. For example, k==0
Constraints	Constraints are placed on model call, states, and transitions to limit the number of test generated, and result in a true or false condition. For example, k<3.
Action	The action field is the place where dynamic context is updated. Actions are preformed when the transition is encountered during test generation. For example, k=k+1.
TEST INFO	In the TEST INFO field, you enter text, macros, or hyperlinked objects so their values are displayed in the resulting testing data.

For example, there are two states in the sub-model WaitsForCardInReader: CardInReader and CardNotInReader. If we wanted to assign likelihood 99.9 and 0.1 to them respectively, just simply enter the figures in fields of Likelihood in the two transitions. Likelihood information should be obtained from the operational profile.

The information for predicates and actions can be found in the SRS. For example, it is stated in the PACS SRS that:

- *In case of failure to read the ID Card, or entering the PIN in 3 tries, an alert is sent to the officer.*

The segment “ 3tries” clearly indicates the existence of a predicate k<3 in the corresponding transitions. In addition, the segment also indicates that every time when the sub-models SwipeCard and EnterPIN are executed, the program should increase the counter by one. This is an action "k=k+1" in the transition between SwipeCard and SwipeCard and EnterPIN and EnterPIN.

The TEST INFO field is used to enter the test script for the test execution tool. Our scripts for WinRunner are typed in the field of TEST INFO.

Variables may be needed to perform some functions in the models. For example, in the model WaitsForCardInReader, we want to set a flag to represent if the card is in the reader. Then a variable is required.

To create variables in a Model, complete the following steps:

4. Locate the Model name in the Project window where the variable(s) will be used, or create a scope variable on the Entry (Root) Model, so it can be used throughout the Project.

5. Right click the mouse on the Model name and choose Add Variable from the drop down menu. The Edit New Variable window is displayed as shown in Figure A-6.

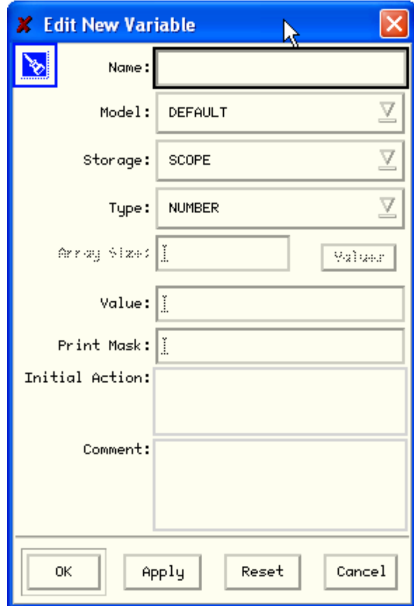

Figure A-6 Edit New Variable

6. Type in the variable name in the Name field.

7. Select a Storage Class from the drop down list box in the Storage field. There are four options: Scope, Local, Param, and Ref Param.

8. Select a data type from the drop down list box in the Type field. There are four data types: NUMBER, NUMBER ARRAY, STRING, and STRING ARRAY.

The variables can be used after they are defined in the model. For example, we used two variables (Prob and CardInReader) in the model WaitsForCardInReader. Variable Prob is used to calculate the probability that a test goes through the path. Variable CardInReader is used as a flag. Applying these variables to the foregoing example, we could modify the model as shown in Figure A-7.

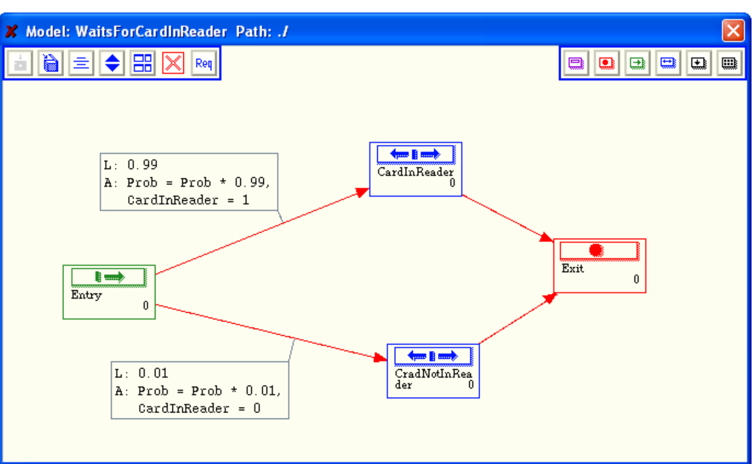

Figure A-7 Transition Edited

In the model, we should also tell the test executor (we use WinRunner as the executor) what to do during the testing. This information is written in the fields of TEST INFO. For example, if we want the executor to wait one second, just type "wait(1)" in the field of TEST INFO in the transition. For more information, please refer to WinRunner's Manual and also to section 3 in this document.

3. Generate Test Cases

Before running the test, we should set some parameters for the testing. To run the test generator, click the Run Mode button, or choose Tests-> Run Settings from the project window. The run settings window is displayed as shown in Figure A-8.

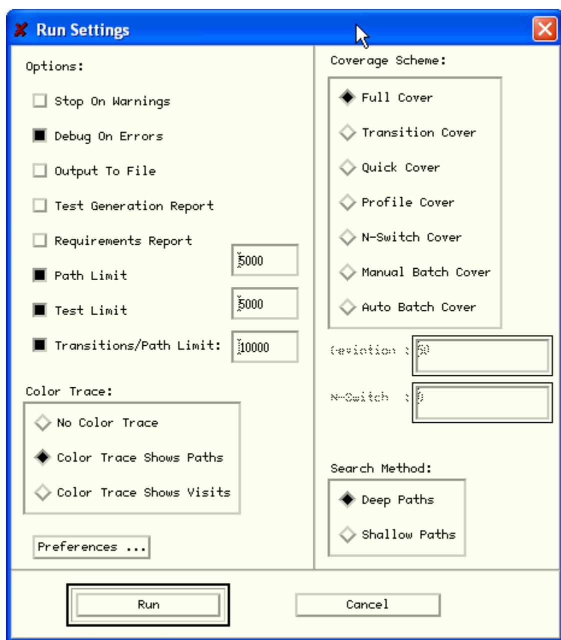

Figure A-8 Run Setting

The major options in the run settings window are described in Table A-4.

Table A-4 Run Settings Window Options

Options	Description
Stop On Warning	Aborts test generation if a run-time warning appears.
Output To File	Creates a file to write the test information. Writes the necessary files for test replay and formatted test file creation.
Test Generation Report	When selected, will produce an report to a default file. Each time the test generator is run, this file will be overwritten.
Requirements Report	When selected, the requirements report will produce a report of how your project met the requirements established in the requirements table.
Test Limit	Sets a limit on the number of tests being generated. Enter the maximum number of tests.
Transitions /Path Limit	Sets a limit on the number of transitions per path. Enter the maximum number of transitions.
Full Cover	Covers all sequence permutations through every state, every

	model and ever transition.
Transition Cover	Ensure that every transition is included in at least one test.
Quick Cover	Generates a minimal set of diverse tests with high model coverage in a relatively short time.
Profile Cover	Generates a user defined number of tests based upon the likelihood information included in the models, and the user specified constraints.
N-Switch Cover	This is a coverage that is between full and transition.
Manual Batch Cover	User specifies which models are high producers then runs the high path producing models in transition/full cover combination in order to present the user with optimal test sets.
AutoBatch Cover	Same as manual batch cover except that the system chooses the high path producers.

We could choose the coverage scheme from this window and set the path limit, test limit and the transitions/path limit. In our case, we will mostly be using the Profile Cover run option. Click Run to run the test generator. After the test case generation, the result is output to the file assigned.

Reference:

[1] *TestMaster Reference Guid*e. 2000, Teradyne Software & System Test: Nashua, New Hampshire.

Bibliography

[1] B. Li, M. Li, and C. Smidts, "Integrating Software into PRA," presented at 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003), Denver, Colorado, USA, 2003.

[2] B. Li, M. Li, and C. Smidts, "Integrating Software into PRA: A Test-Based Approach," *Risk Analysis*, vol. 25, pp. 1061-1077, 2005.

[3] B. Li, M. Li, and C. Smidts, "Integrating software into PRA: Software related failure mode taxonomy," *Risk Analysis*, 2005,under press.

[4] A. Lee, C. Smidts, B. Li, and M. Li, "Validation of a Software-Related Failure Mode Taxonomy," presented at Probabilistic Safety Assessment and Management (PSAM7), Berlin, Germen, 2004.

[5] R.R.Lutz, "Targeting Safety-Rated Errors During Software Requirements Analysis," *ACM SIGSOFT Symposium on Foundations of Software Engineering*, vol. 18, pp. 99-106, 1993.

[6] R. Chillarege, W. L. Kao, and R.G.Condit, "Orthogonal Defect Classification-A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, vol. SE-18, pp. 943-956, 1992.

[7] P.L.Goddard, "Software FMEA Techniques," presented at Annual Reliability and Maintainability Symposium, 2000.

[8] C. Smidts, M. Stutzke, and R. W. Stoddard, "Software Reliability Modeling: An Approach to Early Reliability Prediction," *IEEE Transactions on Reliability*, vol. 147, pp. 268-278, 1998.

[9] C. Smidts and D. Sova, "An Architectural Model for Software Reliability Quantification: Source of Data," *Reliability Engineering and System Safety*, vol. 64, pp. 279-290, 1999.

[10] D. R. Wallace and D. R. Kuhn, "Failure modes in medical devices software: an analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, vol. 18, pp. 351-371, 2001.

[11] C. C. Michael and R. C. Jones, "On the Uniformity of Error Propagation in Software," presented at Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97), 1997.

[12] M. C. Thompson, D. J. Richardson, and L. A. Clarke, "An Information Flow Model of Fault Detection," presented at Proceedings 1st International Symposium on Software Testing and Analysis (ISSTA), 1993.

[13] D. J. Richardson and M. C. Thompson, "An analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection," *IEEE Transactions on Software Engineering*, vol. 19(6), pp. 533-553, 1993.

[14] D. J. Richardson and M. C. Thompson, "The RELAY model of Error Detection and its application," presented at Proc. 2nd Workshop Software on software Testing, Verification and Analysis, Banff, Canada, 1988.

[15] J. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, vol. 18(8), pp. 717-727, 1992.

[16] J. Voas, L. J. Morell, and K. W. Miller, "Predicting Where Faults Can Hide from Testing," *IEEE Software*, vol. 8(2), pp. 41-48, 1991.

[17] J. Voas and K. Miller, "Dynamic Testability Analysis for Assessing Fault Tolerance," *High Integrity Systems Journal*, vol. 1(2), pp. 171-178, 1994.

[18] J. Voas, "Error propagation analysis for COTS System," *Journal of Computing & Control Engineering*, vol. 8(6), pp. 269-272, 1997.

[19] J. Voas, "Software Testability Measurement for Assertion Placement and Fault Localization," presented at Proc. 2nd International Workshop Automated and Algorithmic Debugging (AADEBUG'95), St. Malo, France, 1995.

[20] T. M. Khoshgoftaar, R. M. Szabo, and J. M. Voas, "Detecting program modules with low testability," presented at Proceedings of the International Conference on Software Maintenance, Opio, France, 1995.

[21] B. Murrill, L. Morell, and E. Olimpiew, "A Perturation-based Testing Strategy," presented at Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland, 2002.

[22] M. Hiller, A. Jhumka, and N. Suri, "On the Placement of Software Mechanisms for Detection of Data Errors," presented at Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002), Washington, DC, USA, 2002.

[23] M. Hiller, A. Jhumka, and N. Suri, "An Approach for Analyzing the Propagation of Data Errors in Software," presented at Dependable System and Networks (DSN 2001), Göteborg, Sweden, 2001.

[24] A. Jhumka, M. Hiller, and N. Suri, "Assessing inter-modular error propagation in distributed software," presented at 20th IEEE Symposium on Reliable Distributed Systems, New Orleans, LA, 2001.

[25] D. M. Nassar, W. A. Rabie, M. Shereshevsky, N. Gradetsky, H. H. Ammar, B. Yu, S. Bogazzi, and A. Mili, "Estimating Error Propagation Probabilities in Software Architectures," College of Computer Science, New Jersey Institute of Technology.

[26] D. M. Nassar, W. A. Rabie, M. Shereshevsky, N. Gradetsky, H. H. Ammar, B. Yu, S. Bogazzi, and A. Mili, "A Framework for Error Propagation Analysis of Software Architecture Specifications," presented at 13th IEEE International Symposium of Software Reliability Engineering (ISSRE' 2002), Annapolis, MD,USA, 2002.

[27] W. L. Kao, D. Tang, and R. K. Iyer, "Study of Fault Propagation Using Fault Injection in the UNIX System," presented at Proc. of the 2nd Asian Test Symposium, Beijing, China, 1993.

[28] J. Guan and J. H. Graham, "Diagnostic reasoning with fault propagation digraph and sequential testing," *IEEE Transactions on Systems Man and Cybernetics*, vol. 24(10), pp. 1552-1558, 1994.

[29] A. J. Offutt, "Atomatic test data generation," in *Department of Informtion and Computer Science*, vol. Ph.D: Georgia Institute of Technology, 1988.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, pp. 34-41, 1978.

[31] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. SE-8, pp. 371-379, 1982.

[32] M. Hiller, A. jhumka, and N. Suri, "PROPANE: An Environment for Examining the Propagation of Errors in Software," presented at Proc. Int. Symp. on Software Testing and Analysis (ISSTA '02), 2002.

[33] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," presented at Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA), San Diego, CA, USA, 1996.

[34] S. J. Zeil, "Perturbation Techniques for Detecting Domain Errors," *IEEE Transactions on Software Engineering*, vol. 15(6), pp. 737-746, 1989.

[35] W. G. Cochran, *Sampling techniques*, 3rd ed: John Wiley & Sons, Inc., 1997.

[36] "PACS Requirements Specification," G. Lockheed Martin Corporation Inc., MD, Ed., 1998.

[37] "PACS Design Specification," G. Lockheed Martin Corporation Inc., MD, Ed., 1998.

[38] *TestMaster Reference Guide*. Nashua, New Hampshire: Teradyne Software & System Test, 2000.

[39] *WinRunner TSL Reference Guide*. Sunnyvale, CA: Mercury Interactive Corp., 2002.