1990

# A dynamic failure model for performing propagation and infection analysis on computer programs

Jeffrey Mark Voas
*College of William & Mary - Arts & Sciences*

## Recommended Citation

Voas, Jeffrey Mark, "A dynamic failure model for performing propagation and infection analysis on computer programs" (1990). *Dissertations, Theses, and Masters Projects.* William & Mary. Paper 1539623788.

https://dx.doi.org/doi:10.21220/s2-dd5w-cp84

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road Ann Arbor MI 48106-1346 USA
313 761-4700 800 521-0600

Order Number 9033124

# A dynamic failure model for performing propagation and infection analysis on computer programs

Voas, Jeffrey Mark, Ph.D.

The College of William and Mary, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# A DYNAMIC FAILURE MODEL FOR PERFORMING PROPAGATION AND INFECTION ANALYSIS ON COMPUTER PROGRAMS

A DISSERTATION

PRESENTED TO

THE FACULTY OF THE DEPARTMENT OF COMPUTER SCIENCE

THE COLLEGE OF WILLIAM AND MARY IN VIRGINIA

IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

BY

JEFFREY MARK VOAS

1990

# APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Jeffrey Mark Voas, Author

Approved, March 1990

Larry J. Morell

Larry J. Morell, Dissertation Director

William L. Bynum

William L. Bynum

Sidney H. Lawrence

Sidney H. Lawrence

Keith W. Miller

Keith W. Miller

Paul K Stockmeyer

Paul K. Stockmeyer

# Dedication

This thesis is dedicated to my parents Larry Keith Voas and Sarah Jane Strong Voas, and to my deceased grandfather Allan Clarence Strong for all of their love and encouragement. I thank God for all He has given me.

# Contents

# Acknowledgements

The author would like to express extreme appreciation to Larry J. Morell, for all of the time, support, and ideas that he has provided. Appreciation is also given to the four additional committee members for their willingness to serve on my committee: William L. Bynum, Sidney H. Lawrence, Keith W. Miller, and Paul K. Stockmeyer. The writer would like to express appreciation to Stephen K. Park for his excellence in lecturing and explaining mathematical principles both in the classroom and outside of it. Appreciation is given to P. Kearns and R. Noonan for administrating the Department of Computer Science SUN network which was essential in getting this document together. Appreciation for use of resources is also given to C. Andersen for use of his SUN 3/60 workstation and NEC laser printer. Gratitude is owed to Larry J. Morell and Keith W. Miller for their financial support to the author through research assistantships from NASA/Langley Grants NAG-1-824 and NAG-1-884.

# List of Tables

# List of Figures

## Abstract

This thesis introduces a methodology for determining program locations where faults can easily hide. It is a program structure-based model that analyzes program flow both statically and dynamically; each program location is analyzed relative to its preceding locations and succeeding locations. A statistical model termed ***propagation analysis*** studies the relation between incorrect internal data states and their affect on the output. ***Infection analysis*** is a statistical model which studies the relation between classes of faults and internal data states. Together these two models combine to form one model of analyzing programs termed Propagation and Infection Analysis(PIA).

PIA employs aspects of both software testing methods and verification techniques. The results of PIA distinguish it from traditional verification efforts however. Verification compares a program with its specification. The ultimate goal of verification is to show the program is correct with respect to its specification. PIA characterizes a program in terms of how its failure behavior will be impacted by the presence of faults at various locations. A location which minimally impacts the failure behavior is called ***fault insensitive.*** The goal of PIA is to identify fault insensitive location. Since program correctness, safety, and reliability are all intricately connected to the presence (or absence) of faults in the code, PIA therefore provides information useful in quantifying the effectiveness of other verification activities.

The implementation of the propagation and infection analysis model is performed through dynamic executions of the program. Propagation analysis quantifies the impacts on a program after its internal data states have been altered; infection analysis quantifies the impacts on internal data states that "common" faults have once injected at program locations. The statistics gathered from both altered data states and altered locations are then used to make predictions about expected program behavior if a fault were there.

Models are also provided for applying the results from propagation and infection analysis to a variety of applications dealing with software quality assurance. These include software complexity, debugging, software testing, software reliability, software security, and probable correctness.

# A DYNAMIC FAILURE MODEL FOR PERFORMING PROPAGATION AND INFECTION ANALYSIS ON COMPUTER PROGRAMS

# Chapter 1

# Introduction

This thesis presents both a model and an implementation for quantifying the impact that a particular location in the program has on the program's output behavior. This enables identification of program locations where faults may more easily be hidden. Quantification of the impact of such sections is important in areas such as probable correctness, software reliability, and program testing.

## 1.1 Definition of the Problem

This thesis explores the problem of *determining where a fault can easily hide.* It does this by quantifying the impact that each location has on a program's output behavior. With this knowledge, the complementary information is also revealed: *determining where a fault can not easily hide through the same quantifying process.* This thesis presents a model and an implementation for solving this problem; the method does not use the program's specification or an oracle. The scheme requires no prior knowledge of the software specification, design, or implementation.

The method introduced is called "Propagation and Infection Analysis(PIA)". Propagation and infection analysis employs aspects of both software testing methods and verification techniques [7]. The results of propagation and infection analysis distinguish it from traditional verification efforts however. Verification compares a program with its specification. The ultimate goal of verification is to show the program is correct with respect to its specification [21]. Propagation and infection analysis characterizes a program in terms of how its failure behavior will be impacted by the presence of faults at various locations. A location which minimally impacts the failure behavior is called *fault insensitive.* The goal of propagation and infection analysis is to identify fault insensitive locations. Since program correctness, safety, and reliability are all intricately connected to the presence (or absence) of faults in the code, propagation and infection analysis therefore provides information useful in quantifying the effectiveness of other verification activities.

Determining the impact a fault will have on a program's variables(internal or output) is difficult. Examples can be created where:

1. removing a fault increases the failure rate,
2. adding a fault decreases the failure rate, and
3. adding two identical faults to different locations produce radically different failure rates [22].

This thesis builds on ideas found in [16, 5]. Morell [16] defines a *creation condition* as a boolean condition describing program states in which the substitution of alternate code alters the computation. He further defines a *propagation condition* to be a boolean condition under which two altering states produce different output. These ideas are extended into the methodology of propagation analysis and infection analysis [22]. Propagation analysis empirically determines the probability of a propagation condition being satisfied; infection analysis empirically determines the probability of a creation condition being satisfied.

## 1.2 Importance of the Problem

Quantification of the impact of each location on the program behavior is important in areas such as probable correctness [6], software reliability [17, 2], and program testing [7]. With the information collected from propagation and infection analysis, the following can be determined.

1. Where to get the most benefit from limited testing resources (fault insensitive locations require more testing, therefore by identifying these locations, propagation and infection analysis saves resources which can be applied to the more critical fault insensitive locations; propagation and infection analysis results can save testing locations with more input points than are necessary to be convinced the location is probably correct),
2. Where not to emphasize testing; (propagation and infection analysis results may show extreme fault insensitivity thereby pinpointing locations for which no reasonable amount of testing can be performed to show high confidence in probable correctness, thus alternative techniques should be applied),

3. The degree to which testing must be performed in order to be convinced that a location is probably correct; (propagation and infection analysis results may be used to determined how many test cases are necessary to be convinced a location is probably correct), and

4. Whether or not software should be rewritten (propagation and infection analysis results may be used as a guide to whether software is accepted, i.e., if a piece of software has more than some threshold of fault insensitive locations, then the product may be rejected since too much testing will be required before probable correctness can be shown).

There are other applications of these ideas; several are discussed in Chapter 5. The results of propagation and infection analysis tell where faults are more or less likely to exist, and this points to locations in the code where extra attention is warranted.

## 1.3 Survey of Related Techniques

There exist many schemes for fault detection in computer programs; these schemes are generally classified as either static or dynamic. Two broad schemes of static analysis are proof-of-correctness and program review. Testing, software reliability, and propagation and infection analysis are dynamic analysis schemes. *Proof-of-correctness* is static in that a formal proof is constructed to show that the function computed by the code is correct with respect to the specification [15, 1]. *Program review* inspects both the code and design to reduce the number of errors [3]. *Testing* is dynamic in that the program is executed on actual inputs. *Software reliability* measures the probability of the failure-free operation of software in some time interval [2]. The remainder of this section discusses how propagation and infection analysis is both unique yet related to dynamic and static schemes for analyzing the function a program computes.

Testing plays a significant role in analyzing software for faults. *Statement testing* attempts to execute every statement at least once; *branch testing* is another dynamic structural testing scheme that requires that each branch be executed at least once [15, 8, 19]. Both of these techniques are structural methods of covering software. However executing a location and not observing a failure merely provides one data point for estimating whether or not the statement contains a fault. Propagation and infection analysis is another structural method which covers the software structure. However propagation and infection analysis extends this coverage idea to quantify the impact that a fault at a location will have on the failure rate.

*Mutation testing* [20] is testing strategy which evaluates the input points, by taking a program $P$ and producing $n$ versions(*mutants*) of $P$, $[p_1, p_2, ...p_n]$, that are syntactically different from $P$. If the input points distinguish the mutants from $P$, then it is assumed that if the actual program works with those test cases, the program is good. Mutation testing assumes the "competent programmer hypothesis" which states that a competent programmer produces code that is close to being correct, where "close" means only a few syntax changes are required to correct the program. It also assumes that faults that interact can be caught with test data that reveals single faults, i.e., fault coupling is ignored [14]. Mutation testing tests input data; good test data kills all mutants. Infection analysis uses mutation testing ideas in the following way: Syntactic changes are made to program locations; however, in propagation and infection analysis, these syntactic changes must cause semantic changes. Propagation analysis generalizes the applicability of mutation testing by allowing the data state to be perturbed. Note, however, that PIA's goal is significantly different than mutation testing. Whereas mutation testing tests the input data, infection analysis tests the location's ability to sustain a change in its semantics yet not change the state of the program.

*Error-based testing* attempts to define certain classes of errors and the subdomain of the input space which should reveal any error of that class if that error type exists in the program [18, 23]. Morell [16] proves properties about error-based strategies concerning certain errors that can and cannot be eliminated using error-based testing. Since error-based testing restricts the class of computable functions, it is limited as well. Error-based testing defines errors in terms of their syntax. Propagation and infection analysis is related to error-based testing in that it advances the notion of error-based testing by defining classes of errors in terms of their impact on the state of the program. Hence determination can be made for each location of what a state error of a particular impact has on the program's output behavior. If the output behavior of the program is not similar to the expected impact, then a fault of that particular impact can be dismissed as not occurring. Hence propagation analysis can show that an infinite set of errors of a particular impact is unlikely to exist in the program.

*Fault-seeding* is an error-based technique used to estimate both the number of faults remaining as well as their type. Faults are seeded and the "seeded" version is then run. Based upon the number of faults discovered, an estimate of the number of remaining faults is made [12]. A drawback is that if the seeded faults are not representative of the inherent faults, the estimate is invalid. *Stratified fault-seeding* [13] improves fault-seeding by using the results from propagation analysis and infection analysis. It can be used to show that faults of a particular impact do not exist, instead of a syntactic class of faults.

Testing and propagation and infection analysis are fundamentally different. Testing's goal is to detect faults through the production of failures; propagation and infection analysis's goal is to statistically determine whether or not a location will reveal a fault through the production of a failure. Propagation and infection analysis may be viewed as a method for producing raw information that is central to the analysis of white-box testing strategies.

Software reliability is concerned with program output behavior. It attempts to predict from previous behavior either the probability of no failures in a specified time interval, or the number of remaining faults [2, 17]. In general, software reliability models are black-box schemes which do not consider the code. By relating fault presence to failure tendency, propagation and infection analysis may serve as the basis for a white-box model of software reliability.

Proof-of-correctness is a static method which can show the absolute correctness of a program. There are programs, however, where no such proof is possible. Another problem with correctness proofs is proving the correctness of the proof. Theoretically, Propagation and infection analysis is applicable to any program, thus programs for which proofs-of-correctness are not possible may still be analyzed with propagation and infection analysis. For programs where proofs-of-correctness are not possible, Propagation and infection analysis can provide information on both where and to what degree to perform testing. Proofs of correctness require a specification; propagation and infection analysis does not require a specification.

In summary, propagation and infection analysis contains information not previously available, information that can be used by both dynamic and static verification schemes. The difference between propagation and infection analysis and other methods is the information necessary to use the method and the information provided by the method. Both these differences are significant in several diverse applications.[1]

## 1.4 Propagation and Infection Overview

Propagation and Infection Analysis is built upon the three conditions which are necessary and sufficient for a program to fail:

1. a fault must be reached,

[1] See Chapter 5.

2. the fault must cause an incorrect internal data state, and
3. the incorrect internal data state must eventually cause incorrect output.

Thus for a program location to be fault insensitive, one of the three conditions above must have a high probability of being false for a randomly selected input. A rarely executed location could easily hide a fault, since the location rarely is presented the opportunity to affect the output behavior. Next, suppose there is a location which should have the correct statement x := x mod 10000000. And further suppose the value of x prior to this statement is almost always positive and less than 100. Then there are many constants which can replace $10^7$ and yet still not produce an incorrect internal data state. Hence a fault could easily hide here as well. Finally, suppose a program outputs x*y and y is almost always zero. The location where x is defined is a location where a perturbed data state (where x has a perturbed value) frequently produces correct output. This too is a location where a fault can easily hide.

For each location, propagation and infection analysis statistically estimates for a probability of each of these three conditions occurring. Thus propagation and infection analysis is a white-box scheme for quantifying the impact that each location has on the failure behavior of a program.

## 1.5 Thesis Organization

The organization of the thesis is as follows: defining the area of interest, defining a model for the area of interest and its definitions, developing algorithms for implementing the model, implementing and experimenting with the algorithms, and showing the application of the model. A chapter overview follows:

**Chapter 2: Terminology.** This chapter restates standard definitions from software engineering and provides definitions specific to the propagation and infection analysis model. For those familiar with dataflow analysis[10, 11, 9, 4], the first section may be skimmed.

**Chapter 3: Algorithms for Propagation and Infection.** Chapter 3 presents algorithms for implementing the model introduced in Chapter 2.

**Chapter 4: Propagation and Infection Methodology.** This chapter describes the parts of the PIA methodology and how they interact. It also presents assumptions about the input program. Chapter 4 introduces alternative schemes for implementing the PIA model to those in Chapter 3, which are analytical versus computational.

**Chapter 5: Propagation and Infection Applications.** This chapter presents several application areas of propagation and infection analysis: software reliability, probable correctness, debugging, software testing, and software metrics. A brief introduction or references to each area is given, as well as introductory models showing how propagation and infection analysis may be applied.

**Chapter 6: Empirical Results.** Chapter 6 presents the results of several experiments on non-trivial programs showing the value of the model and implementation. Also, Chapter 6 empirically shows an important characteristic of similar versions of a program: *the propagation and infection results from one version of a program are often identical for another version.* In cases where they are not identical they are often very similar. This means the analysis may begin sooner in the software life-cycle, and the analysis results do not change drastically as faults are removed.

**Chapter 7: Conclusions.** This chapter briefly summarizes the accomplishments of the thesis and acknowledges the weaknesses and limitations of both the model and implementation. Future work to be performed is detailed.

There are four appendices in the thesis. The following briefly summarizes their contents:

**Appendix A: Mathematical Preliminaries.** Appendix A contains introductory material from several areas of computer science and mathematics, including graphs, regular expressions, conditional probability, and Monte Carlo simulation.

**Appendix B: Programs for Propagation and Infection Estimates of qcksrt.** This appendix shows the programs used on a quicksort program to produce the results of the thesis model. It is included to show exactly how estimates are produced.

**Appendix C: Program Used in Experiment I.** This appendix contains the source code of a particular function used in one experiment from Chapter 6.

**Appendix D: Notation and Symbols** Appendix D is a summary of the notation and symbols used throughout the thesis. This appendix will be useful as a quick reference guide during reading.

# References

[1] ALFS BERZTISS AND MARK A. ARDIS. *Formal Verification of Programs SEI Curriculum Module SEI-CM-20-1.0.* Technical Report, Software Engineering Institute, Carnegie Mellon University, 1988.

[2] C. V. RAMAMOORTHY AND F. B. BASTANI. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering SE-8*, pp. 543–371, July 1982.

[3] M. FAGAN. Design and code inspections to reduce errors in program development. *IBM Systems Journal 15*, pp. 182–211, 1976.

[4] LEON J. OSTERWEIL AND LLOYD D. FOSDICK. Dave-a validation error detection and documentation system for fortran programs. *Software Practice and Experience*, pp. 473–486, October-December 1976.

[5] LARRY J. MORELL AND RICHARD G. HAMLET. *Error Propagation and Elimination in Computer Programs.* Technical Report TR-1065, University of Maryland, Department of Computer Science, July 1981.

[6] RICHARD G. HAMLET. Probable correctness theory. *Information Processing Letters*, pp. 17–25, April 1987.

[7] EDWARD MILLER AND WILLIAM E. HOWDEN. *Tutorial: Software Testing and Validation Techniques.* IEEE Computer Society Press, 1981.

[8] J. C. HUANG. An approach to testing. *ACM Computing Surveys*, pp. 113–128, September 1975.

[9] BODGAN KOREL. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9), September 1988.

[10] BODGAN KOREL. The program dependence graph in static program testing. *Information Processing Letters*, January 1987.

[11] LANUSZ W. LASKI AND BODGAN KOREL. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[12] HARLAN D. MILLS. *Software Productivity.* Boston: Little, Brown, 1983.

[13] JEFFREY M. VOAS AND LARRY J. MORELL. *Fault Sensitivity Analysis(PIA) Applied to Computer Programs.* Technical Report WM-89-4, College of William and Mary, Department of Computer Science, December 1989.

[14] L. J. MORELL. A model for code-based testing schemes. *Fifth Annual Pacific Northwest Software Quality Conference*, pp. 309–326, 1987.

[15] LARRY J. MORELL. *Unit Testing and Analysis SEI Curriculum Module SEI-CM-9-1.0.* Technical Report, Software Engineering Institute, Carnegie Mellon University, October 1987.

[16] LARRY JOE MORELL. *A Theory of Error-based Testing.* Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[17] MUSA AND IANNINO AND OKUMOTO. *Software Reliability Measurement Prediction Application.* McGraw-Hill, 1987.

[18] ELAINE J. WEYUKER AND THOMAS J. OSTRAND. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, pp. 236–246, May 1980.

[19] ROBERT L. PROBERT. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, January 1982.

[20] RICHARD A. DEMILLO AND RICHARD J. LIPTON AND FREDERICK G. SAYWARD. Hints on test data selection: help for the practicing programmer. *Computer*, pp. 34–41, April 1978.

[21] STEPHEN R. SCHACH. *Software Engineering.* Aksen Associates Incorporated Publishers, 1990.

[22] LARRY J. MORELL AND JEFFREY M. VOAS. *Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability.* Technical Report WM-88-2, College of William and Mary, Department of Computer Science, September 1988.

[23] STEVEN J. ZEIL. Testing for perturbations of program statements. *IEEE Transactions on Software Engineering SE-9*, May 1983.

# Chapter 2

# Terminology

For the reader who is unfamiliar with software engineering terminology, this chapter gives a brief introduction to particular terms from software engineering that are related to this research. These terms appear throughout the chapters and in many method-related definitions. For those interested in reviewing the requisite mathematics needed, Appendix A contains brief synopses.

This chapter's main purpose is to introduce the definitions specific to the propagation and infection analysis model; some are in [5]. This chapter also contains definitions tailored for particular application areas of propagation and infection analysis which are introduced in Chapter 5. Additional definitions appear in latter chapters as needed. The first section of Chapter 2 gives definitions that uniquely identify code fragments; this is necessary since propagation and infection analysis is a structure-based method. Section 2.2 contains definitions for implementing the PIA methodology on the code "chunks" defined in Section 2.1.

## 2.1 Abstraction Level Definitions

A program may be viewed at various levels of abstraction. The *level of abstraction* refers to the method of grouping program pieces into particular units. Forming levels of abstraction allows for a convenient way to either discuss specific locations, semantically meaningful groups of locations, or syntactically meaningful groups.

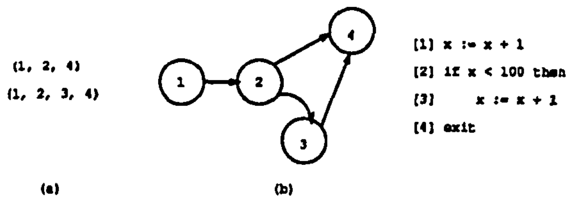
The highest abstraction level is the entire program. The lowest level is one bit of the object program. Other low level choices include the assembly code instructions and machine code instructions. Higher levels come from the source program; choices include a statement, a path, a module, a language sub-construct, a sub-expression, and a collection of paths.

Without losing generality, it is assumed the code is structured. For a program to be structured as defined in [3], all control logic must be handled by (1) simple sequencing, (2) if-then-else, and (3) do-while constructs. There are additional requirements in the definition of *structured programming* in [3] that are not of concern here. A *program* $P$ is a sequence of source statements that compile. The function program $P$ computes is denoted by $[P]$ and this notation is adapted from Mills *et al.*[1]. $[P]$(x) denotes the output $P$ computes on input $x$. A ***specification*** $S$ alone determines correctness for any input/output pair, and $\phi$ is the input domain for $S$. The function computed by $S$ is denoted similarly by $[S]$, and a program $P$ is *correct* if and only if $[S] = [P]$. For $[S] = [P]$ to be true, $\forall x \in \phi \; ([S](x) = [P](x))$ must be true.

A *location* in a program is defined by a language construct or sub-construct in the source program. For instance, if (condition) then, while (condition) do, and assignment statement are each considered as a location. Other high-level language constructs such as case (a) of and repeat (condition) until may also be semantically decomposed into locations according to their functional equivalence using if (condition) then and while (condition) do syntax. To discuss program flow, a digraph is used; graph terminology is presented in Appendix A. The mapping between the digraph(*flowgraph*) and the program depends on the chosen abstraction level; whatever is determined to be a location in the program is represented by a unique vertex in the digraph. A *subflowgraph* is a subgraph of a flowgraph. The flowgraph is created statically. The flowgraph may be augmented to include information at each vertex about what variables are being defined and referenced at that location. The term *node* is used synonymously for vertex.

The node corresponding to the first location in the program is called the *start node* in the flowgraph, and the node corresponding to the last location executed prior to termination is called the *exit node* in the flowgraph. Without loss of generality it is assumed that there is only one start node and one exit node in the flowgraph of the program. A *back arc* is an edge from a higher numbered predecessor as determined by a breadth-first search of the flowgraph, where the root is the start node. Each vertex in the flowgraph can be further classified as either a *begin node*, *intermediate node*, or *finish node*. A node may be both a begin node and a finish node. A *begin node* is either the start node, or a node whose predecessor has a back arc going into it (the predecessor). A *finish node* is either the exit node or a node with a back-arc leaving it. An *intermediate node* is any node which is neither a begin nor a finish node.

A *path* through a digraph is the sequence of nodes encountered in the flowgraph including the start node and the exit node. A path may not be empty. A



```
[1] x := x + 1
[2] if x < 100 then
[3]     x := x + 1
[4] exit
```

Fig. 2.1: Two representations of code, (a) regular expressions (b) flowgraph

*subpath* is any contiguous sequence of endpoints reached on a path. A subpath may not be empty. A path is a subpath of itself. Sets of paths can be represented by regular expressions where the alphabet is the set of vertices and each concatenation of symbols from the alphabet represents a path. Figure 2.1 illustrates a mapping between code, flowgraph, and regular expressions. The notation (1, 3, 5, 4, 2, 4) represents the subpath through the program where location 1 is executed first, then 3, 5, and continually until location 4 is reached a second time. A location in the program is termed *reachable* if and only if there is a subpath to the corresponding vertex from the start node.

A *route* is a subpath starting at a begin node and ending at

1. the first finish node encountered from the begin node, or
2. the next begin node encountered,

where no edge in the subpath is a back-arc. A *trip* represents a path or set of paths whose regular expression contains neither alternation nor reflexive transitive closure, but may have transitive closure for loops allowing for one or more iterations.[1] Also, any edge encountered between endpoints on the trip may not be a back-arc. Not allowing a trip to contain reflexive transitive closure ensures that an indefinite loop or definite loop of zero iterations is categorized differently from a loop with one or more iterations of the body.

A program's set of trips is all paths through its flowgraph; it is a collection of path equivalence classes that associates paths with similar locations. A *subtrip* represents a subpath or set of subpaths whose regular expression contains no

[1] The definition for trip is non-specific in the situation of a loop, since the regular expressions 1 $(2, 3, 1)^+$ and $(1, 2, 3)^+$ 1 satisfy the definition for Figure 2.2 (a). Although both are correct, they represent the same trip. Therefore two regular expressions can represent the same trip, however the $T$ should only include one regular expression per trip.

alternation; the regular expression may have no reflexive transitive closure, but may have transitive closure for loops with one or more iterations. A trip is a subtrip of itself. A ***trip set*** represents one or more trips and is represented by a regular expression that contains alternation. This is helpful when a loop contains a branching statement since the set of trips in this situation combinatorically explodes. The notion of a trip set groups similar trips grouped together. The notation for these two sets is as follows: $TS$ is the set of regular expressions of all trip sets of a program, and $T$ is the set of regular expressions of all trips of the program, where a path represented by a regular expression of a member of $T$ is not represented by a member of $TS$. Likewise, no two elements of $T$ can represent the same path. Thus the elements in $T \cup TS$ represents all paths through the program, and the intersection of paths represented in $T$ and in $TS$ is empty; each regular expression in $T \cup TS$ represents what is termed a ***path equivalence class.*** The set of all path equivalence classes through a program is denoted as $PEC$ ($PEC = T \cup TS$).

Graph (b) in Figure 2.2 shows an instance of why the trip set definition is necessary. Notice in Figure 2.2 and succeeding flowgraph figures in the thesis that a shaded node represents a start node. Consider only several of the regular expressions for the many trips that potentially exist when the loop is executed: (1, 2, 3, 5, 1), (1, 2, 4, 5, 1), (1, 2, 3, 5, 1, 2, 4, 5, 1), and (1, 2, 4, 5, 1, 2, 3, 5, 1). By using the trip set notion, this infinite set of trips my be summarized by the regular expression $(1\ (2, 3, 5 \mid 2, 4, 5))^{+}1$.

### 2.1.1 The Four Abstraction Levels

This section identifies four abstraction levels at which the input program may be viewed. Propagation analysis and infection analysis strictly use the first level, however later application models will use higher levels.

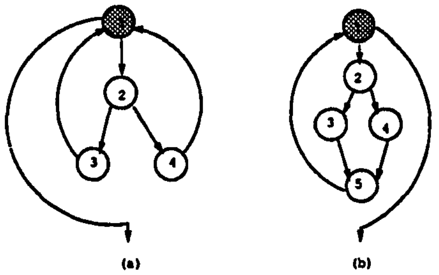
Propagation analysis and infection analysis use the ***statement*** level(Level 1). The rule used to determine whether a "chunk" of code is at level 1 is:

**Rule 2.1** *Any assignment statement or predicate that affects the value of the* pc *is considered as a location in the level 1 flowgraph.*

For instance, for the construct if $(\alpha)$ then $\beta$ else $\gamma$, the predicate if $(\alpha)$ then would be a level 1 location, and each successive location containing a

1. if (condition) then,

2. while (condition) do, or



```
[1] while a <> b do
[2]  if a > b then
[3]    a := a - 1
[4]  else b := b - 1;
```

```
[1] while c < a do
begin
[2]  if b <> a then
[3]    c := c + 1
[4]   else c := c + 2;
[5]  b := b + 1;
end
```

Fig. 2.2: Level 1 flowgraph

3. assignment statement

in the two branches($\beta$,$\gamma$) are considered as level 1 locations. Since the else represents else if not($\alpha$), it is not considered as a location. Figure 2.2 shows if and while constructs being mapped into flowgraphs.

All levels defined above the level 1 are defined for application areas of propagation and infection analysis. The second level is the *route* level(Level 2); it is not mentioned beyond this chapter. It is useful, however, for discussing loops and may be useful in a modular propagation and infection analysis model; this thesis presents a non-modular model. At level 2, there is a subflowgraph for each route. For Figure 2.2's graph (a) there are four routes: (1,2), (2, 3), (1, continuation), and (2, 4).

In Figure 2.2 (a), nodes 1 and 2 are begin nodes; 3 and 4 are finish nodes. There are no intermediate nodes. The next higher level is the *trip* level(Level 3). In Figure 2.2 (a), the trips include (1, 2, 3)$^+$1 and (1, 2, 4)$^+$1(node 1 is used in this example as the exit node instead of creating a dummy node as the exit node). The highest level(Level 4) is the *trip set* level. In Figure 2.2 (a), the trip set is (1, 2, 3 | 1, 2, 4)$^+$1 and for Figure 2.2 (b), the trip set is (1, 2, 3, 5 | 1, 2, 4, 5)$^+$1. For Figure 2.2 (b), the routes are (1, 2), (1, continuation), (2, 3, 5), and (2, 4, 5), and the number of trips is potentially infinite. The begin nodes are 1 and 2, 5 is a finish node, and 3, and 4 are intermediate nodes.

Levels 1 and 2 are termed *subpath* levels; 3 and 4 are termed *path* levels.

Additional examples showing mappings from various constructs to the various abstraction level regular expressions are found in Figure 2.3.

## 2.2 Model and Implementation Definitions

An *error* is a mental mistake made by the programmer, and a *fault* is the manifestation of this mistake in the program. An error can be misreading the specification, typing incorrectly, misunderstanding the algorithm, etc. A fault can be omitted code, an incorrect predicate, wrong variable substitution, incorrect expression, etc. A *fault point* is the location where the fault resides. A *distributed fault* encompasses more than one location. For this model, a fault point is considered as unique, thus distributed faults are not considered. A *failure* occurs when the output differs from the correct value or there is no output when there should be. *Fault size* of a fault at a particular location is the proportion of inputs based on the input distribution that reach the fault and result in failure. Thus for a particular location $z$, if $k$ is the cardinality of the input space and $y$ points fail for some fault $f$, then the fault size of $f$ at $z$ is $\frac{y}{k}$.

A *use* of a variable $v$ is any location $x$ in which $v$ is referenced. A *definition* of a variable $v$ is any location $x$ in which $v$ is assigned a value. Definition can occur via an assignment or input statement. Location $X$ is *data dependent* on location $Y$ iff there exists a variable $u$ such that

1. $u$ is defined at $Y$,
2. $u$ is used $X$, and
3. there exists a control path from location $Y$ to location $X$ along which $u$ is not redefined [2].

*Control dependence* may be defined for the two instructions while and if as follows:

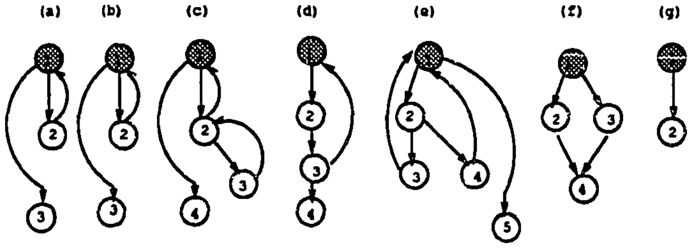
1. if Y then A1 else A2, location $X$ is control dependent on location $Y$ if location $X$ is referenced in A1 or A2,
2. while Y do B, $X$ is control dependent on location $Y$ if location $X$ is referenced in B [2].

[2] Although this is functionally equivalent to i := 1; while i $\leq \alpha$ do $\beta$, the initialization and the control are left together.

[3] Since this can be implemented using while-do it is allowed in the structured language.

| | *(a)* | *(b)* | *(c)* | *(d)* | *(e)* | *(f)* | *(g)* |
|---|---|---|---|---|---|---|---|
| (1) | while $\alpha$ do | for i:=1 to $\alpha$ do[2] | while $\alpha$ do | repeat[3] | while $\alpha$ do | if $\alpha$ then | stmt$_1$ |
| (2) | $\beta$ | $\beta$ | while $\beta$ do | $\beta$ | if $\beta$ then | $\beta$ | stmt$_2$ |
| (3) | etc. | etc. | $\gamma$ | until $\alpha$ | $\gamma$ | else $\gamma$ | |
| (4) | | | etc. | etc. | else $\nu$ | etc. | |
| (5) | | | | | etc. | | |

(a) Code fragments



(b) Corresponding flowgraphs

| *(a)* | *(b)* | *(c)* | *(d)* | *(e)* | *(f)* | *(g)* |
|---|---|---|---|---|---|---|
| 1,3 | 1,3 | 1,4 | $(1,2,3)^+4$ | 1,5 | 1,2,4 | 1,2 |
| $(1,2)^+1,3$ | $(1,2)^+1,3$ | $((1,2)^+(3,2)^+)^+1,4$ | | $(1(2,3 \mid 2,4))^+1,5$ | 1,3,4 | |
| | | $(1,2)^+1,4$ | | | | |

(c) Trips and Trip Sets

| *(a)* | *(b)* | *(c)* | *(d)* | *(e)* | *(f)* | *(g)* |
|---|---|---|---|---|---|---|
| 1,2 | 1,2 | 1,4 | 1,2 | 1,2 | 1,2,4 | 1,2 |
| 1,3 | 1,3 | 1,2 | 2,3 | 2,3 | 1,3,4 | |
| | | 2,3 | | 1,5 | | |
| | | | | 2,4 | | |

(d) Routes

Fig. 2.3: Additional abstraction examples

A *program dependence digraph* can be created with vertices representing locations and arcs representing the control and data dependencies [2]. A dependence digraph differs from the flowgraph since it is a function of the control and data dependencies, i.e., the semantics; the flowgraph is a function of the syntax. By back-chaining through the program dependence digraph from the output location, all locations upon which the output is either data or control dependent can be determined; variables which may affect the output are determined.

A variable $a$ is *directly influenced* by variable $b$ if the location where $a$ is defined is data dependent on the location where $b$ is defined. A variable $a$ is *indirectly influenced* by variable $b$ if there exists a variable $c$ which is directly influenced by variable $b$, and variable $a$ is in directly influenced by variable $c$. Whether $a$ directly influences $b$ or $a$ indirectly influences $b$, it is said that *$a$ influences $b$*. A variable is *active* at a particular location if its value at that location may potentially affect either program control flow or computation in a way which subsequently affects the output. Formally, let $G$ represent the program dependence digraph, and let $G^*$ denote the reflexive transitive closure of $G$. And let $(a, b) \in G^*$ denote that there is a subpath in $G$ from $a$ to $b$. Thus $x$ is an active variable at location $z$ if and only if $(z, O) \in G^*$, where $O$ is a location which produces output, i.e., the location represented by the exit node. So the same variable may be active at one location yet not active at another. The program counter is always treated as a variable and is considered to be active. A variable which is not active is termed *dead*.[4]

A location's *data state* is a mapping of all variables to values which are defined for the executing program. For this model, this state occurs between locations, since the execution of a location is considered to be an atomic operation. Although most source locations are translated into many machine operations which are not atomic, for simplicity, atomic source locations are assumed. $\mathcal{B}_{l,i}(x)$ represents the data state encountered prior to executing location $l$ on the i$^{th}$ iteration from input $x$. $\mathcal{A}_{l,i}(x)$ represents the data state produced after executing location $l$ on the i$^{th}$ iteration from input $x$. Let $\bigcup_{i=1}^{n} \mathcal{B}_{l,i}(x) = \mathcal{B}_l(x)$ and let $\bigcup_{i=1}^{n} \mathcal{A}_{l,i}(x) = \mathcal{A}_l(x)$. If location $l$ is not in a loop and $l$ is reached by all input points, then $\forall x \mid \mathcal{B}_l(x) \mid= 1$.

A *value distribution* at a location for a particular active variable is the distribution of values that occur from the data states at that location from a particular input distribution. A *data space* at a location is the collection of value distributions for all active variables succeeding location $l$. If more than one path or path equivalence class goes through a location, the data space may be thought of parti-

[4] The notion of determining whether a variable is *dead* is similar to dead-code detection in compiler optimization. A variable definition is dead-code iff there is no succeeding subpath to an output location, i.e., not active for any succeeding subpath.

tioned into a ***subdata space*** corresponding to each path or path equivalence class, since each path or path equivalence class may have different sets of active variables at a location with different value distributions. If there are infinitely many paths, then there are infinitely many subdata spaces; hence paths are grouped according to path equivalence classes to create finitely many subdata spaces. Figure 4.2 shows an example where the data space prior to location $x$ is made from two subdata spaces. Figure 2.4[5] shows an example of the value distributions at the data spaces of two locations in the path of some program. A data state that is created for performing propagation analysis or infection analysis by selecting values from the value distributions is termed a ***sampled data state.*** The variables essential in a sampled data state that must be assigned values are the active ones.

A ***virus*** is an incorrect value in a pairing at some data state where correctness is determined by an assertion for that location for that program. The term virus may seem inappropriate in this context, however there is a closer relationship between the virus termed here and the physical virus than between the popularized term virus, such as the Internet worm [4]. The Internet worm was bent on destruction; a biological virus does not necessarily do harm; neither does a virus in a data state. A data state may contain more than one virus. The number of viruses in data state $x$ of location $l$ is termed the degree of the virus and is notated as $\mathcal{D}_l(x)$. This is referred to as the ***error degree*** of the data state. The means by which a data states becomes infected when determining $\mathcal{D}_l(x)$ is arbitrary; whether from an actual fault or other method. It may also be the case that a fault at a location causes several viruses at that location's data space. If a virus exists, the internal data state at that point is termed ***infected.*** If $\mathcal{D}_l(x) = 0$, the data state is termed ***clean.*** A variable is termed ***infected*** if it contains a virus. A variable in a data state is termed ***clean*** if it is not infected.

A ***semantic alternative*** is a copy of the code at a location that has been syntactically altered with the additional restriction that it is semantically different, i.e., [original location] $\neq$ [altered location]. A ***perturbation function*** is a mathematical function that produces "altered" values for variables. In general, it uses the current value of a variable as input and produces an infected value as output.

The ***execution rate*** of a given fault is simply the probability that a random input (from the assumed input distribution) will arrive at the location. It is the following probability:

$$\Pr[\mathcal{B}_l(x) \neq \emptyset] = \Pr[n \neq 0] \tag{2.1}$$

where $n$ is the number of arrivals at location $l$ on random input $x$. So this is just the probability that $n$ does not equal zero. This makes the execution rate a

[5] Figure 2.4 done by Christoph Michael.

function of the input distribution and the fault point.

Let $f_l$ denote the function that is computed at the fault location $l$ and $f_l'$ denote the function that should be computed at the fault location. If for some $y \in \mathcal{B}_l(x)$, $f_l(y) \neq f_l'(y)$, it is said the fault at location $l$ has *infected* $\mathcal{A}_{l,i}(x)$ for the $i$ corresponding to $y$. The *infection rate* of a fault at location $l$ is the conditional probability that a succeeding data state will be infected given that the execution reaches location $l$ for a random input $x$. The infection rate is:

$$\Pr[\mathit{infected}(l,x) \mid \mathcal{B}_l(x) \neq \emptyset] \tag{2.2}$$

where

$$\mathit{infected}(l,x) = \begin{cases} \mathsf{T} & \text{if } \exists y \in \mathcal{B}_l(x)\ f_l(y) \neq f_l'(y) \\ \mathsf{F} & \text{otherwise} \end{cases}$$

The infection rate is a function of the fault and the set of data states that can be present at the fault location.

The process of estimating this probability (equation 2.2) is a function of the fault inserted at the location, the value distributions in the preceding data space, and the method of selecting a data state for that location. The value produced is termed an *infection estimate*, and it estimates infection rates for a class of faults rather than a single fault.

The *propagation rate* of a fault at location $l$ is the conditional probability that the program will fail, given that the fault at $l$ has infected one of its successor data states. The propagation rate is:

$$\Pr[\text{program fails on input x} \mid \mathit{infected}(l,x)] \tag{2.3}$$

The propagation rate of a fault is a function of the set of infected data states created by the fault and the succeeding locations. Propagation rate estimation for a location $x$ is a function of the data states used, the method of infecting the data states, and the succeeding locations that the infected data state reaches. This value produced is termed a *propagation estimate.*

Let $ds_{\mathcal{S}_l} = \bigcup_x \bigcup_i \mathcal{A}_{l,i}(x)$ represent the data space succeeding location $l$ over many input points $x$, and let $ds_{\mathcal{P}_l} = \bigcup_x \bigcup_i \mathcal{B}_{l,i}(x)$ denote the preceding data space of location $l$ over many input points $x$. Then $ds_{\mathcal{S}_{exit}}$ denotes the data space succeeding the exit location of the program and $ds_{\mathcal{P}_{exit}}$ denotes the data space preceding the exit location. Denote the set of inputs that are represented by clean data states at data space $l$ as $\{ds_{\mathcal{S}_l}$ is clean$\}$, i.e., the set of input points that have passed through successor locations of location $l$ and have managed not to have their corresponding data states infected. The set of input points that have produced an infected data state in data space $ds_{\mathcal{S}_l}$ is denoted as $\{ds_{\mathcal{S}_l}$ is infected$\}$. Two different schemes are presented in this thesis for estimating the propagation rate: *failure propagation estimation* and *viral propagation estimation*. The *failure propagation rate* for location $l$, $\mathrm{F}_l$, is the probability that an infected data state at location $l$ causes "differing output". Formally it is

$$\mathrm{F}_l = \Pr[\{ds_{\mathcal{S}_{exit}} \text{ is infected}\} \mid \{ds_{\mathcal{S}_l} \text{ is infected}\}] =$$

$$\frac{\Pr[\{ds_{\mathcal{S}_{exit}} \text{ is infected}\} \cap \{ds_{\mathcal{S}_l} \text{ is infected}\}]}{\Pr[\{ds_{\mathcal{S}_l} \text{ is infected}\}]}. \tag{2.4}$$

The *viral propagation rate* for location $l$, $\mathrm{V}_l$, is the conditional probability that given an infected data state occurs in the preceding data state of location $l$ on some iteration $i$, $\mathcal{B}_{l,i}(x)$, the infection propagates to the succeeding data state of location $l$, $\mathcal{A}_{l,i}(x)$. Formally it is

$$\mathrm{V}_l = \Pr[\{ds_{\mathcal{S}_l} \text{ is infected}\} \mid \{ds_{\mathcal{P}_l} \text{ is infected}\}] =$$

$$\frac{\Pr[\{ds_{\mathcal{P}_l} \text{ is infected}\} \cap \{ds_{\mathcal{S}_l} \text{ is infected}\}]}{\Pr[\{ds_{\mathcal{P}_l} \text{ is infected}\}]}. \tag{2.5}$$

Two different scenarios that may occur to an infected data state once the data state becomes infected are defined in the thesis: *type I cancellation* and *type II cancellation*. *Cancellation* is said to have occurred at location $l$ on the $i^{th}$ iteration of location $l$ if

1. (type I): $\exists i\ \exists x\ \mathcal{D}_l(\mathcal{A}_{l,i}(x)) < \mathcal{D}_l(\mathcal{B}_{l,i}(x))$, or
2. (type II): $\exists i\ \exists x\ \mathcal{D}_l(\mathcal{A}_{l,i}(x)) = \mathcal{D}_l(\mathcal{B}_{l,i}(x))$, where the variable receiving the action at location $l$ is clean in $\mathcal{B}_{l,i}(x)$ yet influenced at location $l$ by an infected variable in data space $\mathcal{B}_{l,i}(x)$.

As an example, suppose every data state in $ds_{\mathcal{S}_l}$ and $ds_{\mathcal{P}_l}$ is infected; then the viral propagation estimate for location $l$ is 1.0. Now suppose that every data state in $ds_{\mathcal{P}_l}$ is infected, however not every data state in $ds_{\mathcal{S}_l}$ is infected; then the viral propagation estimate for location $l$ is $< 1.0$, hence type I cancellation occurred. Type I cancellation occurs when the code at a location removes the infection caused at another location, i.e., the successor location either partially or totally corrects the infection caused by the fault. A special case of type I cancellation occurs when

1. location $l$ is the first location of a branch, thus location $l$'s predecessor location is a conditional location, and

2. on the branch where $l$ occurs, a particular variable or set of variables are no longer active(however they are active on a different branch stemming from $l$'s predecessor location).

As an example, suppose that for a conditional location there are two branches, and location $l$ is the first location on the first branch and location $l'$ is the first location on the second branch succeeding the conditional location. For this special case of type I cancellation,

$$\forall x\ \exists i\ \exists l\ \mathcal{D}_{pred(l,i,x)}(\mathcal{A}_{pred(l,i,x),i}(x)) > \mathcal{D}_l(\mathcal{B}_{l,i}(x)),$$

however

$$\forall x\ \exists i\ \exists l'\ \mathcal{D}_{pred(l',i,x)}(\mathcal{A}_{pred(l',i,x),i}(x)) = \mathcal{D}_{l'}(\mathcal{B}_{l',i}(x)),$$

where $pred(l, i, x)$ denotes the predecessor location of location $l$ on input $x$ on the i$^{th}$ iteration of $l$.[6] When considering type I cancellation in the thesis, this strange phenomenon of type I cancellation is not considered. Type II cancellation is also only defined in Chapter 2; it is not further discussed.

## 2.3 Conclusions

Chapter 2 has provided an overview of the notation and definitions of the thesis. Two of the basic probabilities which are central to the thesis are presented: the infection rate and the propagation rate. Appendix D contains an overview of the notation and symbols from this chapter and the remaining thesis for quick reference.

[6] Similarly throughout the thesis, $succ(l, i, x)$ denotes the successor location of location $l$ on input $x$ on the i$^{th}$ iteration of $l$.

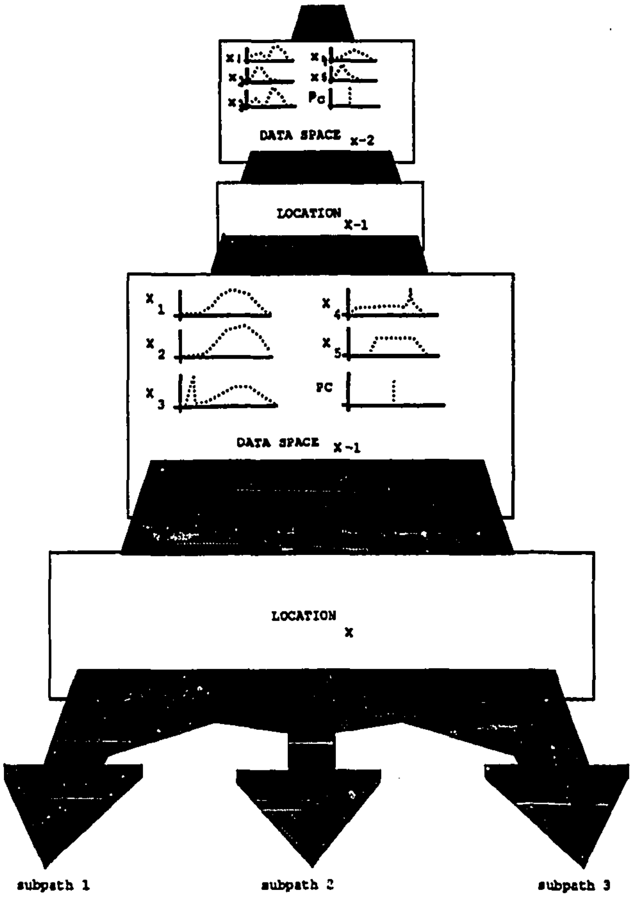


Fig. 2.4: Data space value distributions

# References

[1] H. Mills and V.Basili and J. Gannon and R. Hamlet. *Principles of Computer Programming: A Mathematical Approach.* Allyn and Bacon, 1987.

[2] Bodgan Korel. The program dependence graph in static program testing. *Information Processing Letters*, January 1987.

[3] Harlan D. Mills. *Software Productivity.* Little, Brown, and Company, 1983.

[4] Eugene H. Spafford. Crisis and aftermath. *Communications of the ACM*, 32(6), June 1989.

[5] Larry J. Morell and Jeffrey M. Voas. *Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability.* Technical Report WM-88-2, College of William and Mary, Department of Computer Science, September 1988.

# Chapter 3

# Algorithms for Propagation and Infection

Propagation analysis and infection analysis perform both static and dynamic analysis on the input program. Once completed, the PIA methodology produces two sets of estimates:

1. estimates of probabilities for "if a fault occurs, will a variable or the program counter contain an incorrect value?" and
2. estimates of probabilities for "if an incorrect value occurs, will the program fail?"

These estimates of probabilities are related to the three conditions which are necessary and sufficient for a program to fail:

1. a fault must be reached during execution,
2. the computation at that point must be adversely affected, and
3. the adverse computation must cause incorrect output [6].

*Propagation analysis* [6] is the portion of PIA that attempts to quantify the answer to the second question above. By determining the impact of incorrect internal values on program output, propagation analysis estimates where faults may hide more easily. Propagation analysis estimates this probability by comparing outputs from perturbed computations versus non-perturbed computations of the same program being analyzed; propagation analysis therefore requires no oracle. With propagation estimates and knowledge of the input distribution, "guesses" of where an erroneous internal data state can lie "silent" (without exposing itself) for periods of testing time or operational time are thus discovered.

Propagation analysis performs its task by studying the peculiarities of the code's structure while measuring the code's resistance to erroneous data states, i.e., how much can the variables' values diverge and still produce the same output. It simulates the impact that a fault from the set of *potential faults* may have on a data state; this is accomplished by changing the values of variables at locations in the program. There is no probability associated with whether it is likely that a particular variable might have an incorrect value at a particular location during propagation analysis; it is considered equally probable that any particular variable is incorrect at any location. For instance, if variable $z$ is defined at 100 preceding locations and variable $y$ is defined once, although it is more likely $z$ will be incorrect, no weights are attached in the methodology to enforce this intuition.

Infection analysis [6] is the portion of PIA that quantifies the answer to the first question; it is similar to propagation analysis in that it requires no oracle, and locates where faults may easily hide by determining which faults have low impacts on data states. Infection analysis is performed at each location $l$ ; it measures location $l$'s ability to contain a semantic alternative from a fault class $\mathcal{F}_l$ yet not produce a virus. Each element in $\mathcal{F}_l$ is viewed as equally probable; there are no weights associated with whether a semantic alternative is more or less likely than another semantic alternative.

A particular semantic alternative or erroneous data state may affect one program entirely different than another. In fact, the impact a semantic alternative or erroneous data state has when located at different places within the same program may greatly differ. It is the semantics of the succeeding and preceding locations reached combined with the input distribution that determine the impact of semantic alternatives and injected viruses on a program's behavior.

Sections 3.1, 3.2, and 3.3 present the algorithms for finding infection estimates, viral propagation estimates, and failure propagation estimates. Throughout this thesis, the term *estimate* is meant as a point estimator, $\overline{Y}$, of a some target parameter of interest. The point estimate that is used is the sample mean,

$$\overline{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i,$$

where there are $n$ samples and

$$Y_i = \begin{cases} 1 & \text{event of interest occurred} \\ 0 & \text{otherwise} \end{cases}$$

[7].

## 3.1 Infection Estimation Implementation

The *infection estimate* for a location $l$ is an estimate of the conditional probability that a fault (semantic alternative) existing at location $l$ produces a virus in the succeeding data state of $l$ given that the fault (semantic alternative) is reached. For every location in the input program, there is an infection rate for a particular fault at that location. Our estimate for the infection rate using Chapter 2's notation for a location $l$ with input $x$ where $a \in \mathcal{F}_l$ is:

$$I_{l,a} = \Pr[\mathit{infected}'(l,a,x) \mid \mathfrak{B}_l(x) \neq \emptyset] \tag{3.1}$$

where

$$\mathit{infected}'(l,a,x) = \begin{cases} \mathsf{T} & \text{iff } \exists y \in \mathfrak{B}_l(x)\ f_l(y) \neq f_a(y) \\ \mathsf{F} & \text{otherwise} \end{cases}$$

and $f_a$ is the function computed by semantic alternative $a$. The true infection rate for a particular known fault is found by equation 2.1 and is a function of the fault versus the correct location. The infection estimate is a function of a set of semantic alternatives versus the current location. This is the main difference between the infection rate and the infection estimate. The algorithm for the infection estimate is a function of a class of faults $\mathcal{F}_l$ since not every potential fault at a location can be tried.[1] The algorithm for implementing equation 3.1(as shown in Figure 3.1) is:

1. create a semantic alternative from $\mathcal{F}_l$ for location $l$; call this semantic alternative $a$,
2. set variable count to 0,
3. present the original code at $l$ and the semantic alternative $a$ with a data state from the preceding data space of $l$, and execute both locations in parallel; set a time limit for termination of the semantic alternative, and if execution is not finished in that time interval, assume an infinite loop occurred,
4. compare the resulting data states of $a$ and $l$, and increment count when an infection occurs; repeat steps 3 and 4 $n$ times,

[1] The definition of $\mathcal{F}_l$ is detailed in Chapter 4.

5. divide count by $n$ yielding an infection estimate for semantic alternative $a$, denoted as $I_{l,a}$[2],

6. perform the previous five steps on each element of $\mathcal{F}_l = \{a_1, a_2, ...\}$ creating infection estimates, $I_{l,a_1}, I_{l,a_2}, ...$ for location $l$,

7. several applications[3] of PIA require the minimum infection estimate of a location; to find the minimum, select the lowest infection estimate as the overall infection estimate for location $l$:

$$\min_{a \in \mathcal{F}_l} \{I_{l,a}\} = \min_{a \in \mathcal{F}_l} \{\Pr[\mathit{infected}(l, a, x) \mid \mathcal{B}_l(x) \neq \emptyset]\} \tag{3.2}$$

Omitted location faults have infection estimates close to 1.0 since most locations either have a computational effect or control flow effect. Faults of omitted locations should not affect the infection estimate selected by equation 3.2.

Infection analysis can be partially automated by a system that performs syntax mutation provided that the system produces semantic alternatives versus pure syntactic mutants. It is necessary to provide information about $\mathcal{F}_l$ to such a system so that the semantic alternatives represent $\mathcal{F}_l$ for that location. One example of a system that performs syntax mutation is Mothra [1, 5].

## 3.2 Propagation Estimation Implementation

PIA uses only failure propagation estimates. The algorithm for determining a failure propagation estimate is specific to a location; viral propagation estimates, however, are determined specific to a path. Both estimates are found in a similar manner: from execution-based data.

### 3.2.1 Failure Propagation

The *failure propagation estimate* for a location is an estimate of the conditional probability of a pseudo-failure occurring given that a virus exists in the data state immediately following the location. A *pseudo-failure* occurs when output is

[2] The infection estimate is a point estimator which is the sample mean of the number of infections, however to determine the confidence intervals in the estimate, the confidence interval of the frequency distribution for 95% confidence can be found with $p \pm w$, where $w = 2 \cdot \sqrt{(p \cdot (1-p)/n)}$ and $p =$ count $/n$ [4].

[3] Detailed in Chapter 5.

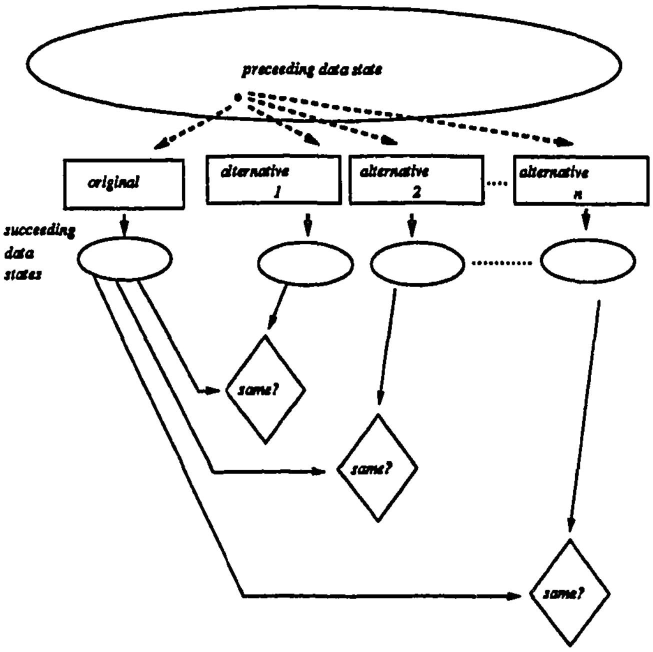


Fig. 3.1: Infection estimation

produced which is different than that which would have occurred had the virus not been injected.[4] Pseudo-failure is determined by the input program, which may or may not be the correct program. For every location in the program, there is a failure propagation rate for a specific fault relative to the correct program as defined by equation 2.2. Our failure propagation estimate for location $l$ and input $x$ is defined by the following conditional probability:

$$\mathrm{F}_{l,a} = \Pr[\text{pseudo-failure on input } x \mid \textit{perturbed}(l, a, x)]. \tag{3.3}$$

where

$$\textit{perturbed}(l, a, x) = \begin{cases} \mathrm{T} & \text{iff } a \in f_l(\mathcal{B}_{l,1}(x)) \text{ and } \textbf{function } \mathsf{perturb(a)}^3 \text{ has executed} \\ & \text{on input } x \\ \mathrm{F} & \text{otherwise} \end{cases}$$

and where $\mathsf{A}_l$ is the set of active variables at location $l$ regardless of the succeeding subpath. Once again, the propagation rate is a function of the correct program,

[4] This was referred to as "differing output" in Chapter 2.

whereas the failure propagation estimate is a function of the current program. One method for estimating the failure propagation estimate of a location $l$ is:

1. Take the set $\mathbf{A}_l$ and select some active variable denoted as $a$,
2. Set variable count to 0,
3. Sample from the distribution of values for $a$ at $l$ in $\mathcal{A}_{l,1}(x)$; it is necessary to sample for values of all active variables at that location so that there is a complete data state from which to start execution; repeat this step $n$ times,
4. Alter the sampled value of $a$ with a perturbation function(defined in Section 3.3), and execute the succeeding code on both the perturbed and original data states,
5. For each observed different outcome between the perturbed data state and the original data state, increment count; increment count if an infinite loop occurs; repeat steps 3-5 $n$ times,
6. Divide count by $n$ yielding a failure propagation estimate, $\mathrm{F}_{l,a}$[5],
7. Repeat the previous six steps on each $a_i \in \mathbf{A}_l$, yielding many failure propagation estimates, $\mathrm{F}_{l,a_1}$, $\mathrm{F}_{l,a_2}$,.... Each active variable is chosen to account for the impacts of omitted location faults,
8. Several applications of PIA require the minimum failure propagation estimate of a location:

$$\begin{aligned}&\min_{a \in \mathbf{A}_l}\{\mathrm{F}_{l,a}\} = \\ &\min_{a \in \mathbf{A}_l}\{\Pr[\text{pseudo-failure on input } x \mid \textit{perturbed}(l,a,x)]\}.\end{aligned} \tag{3.4}$$

Failure propagation estimation may be automated in a similar manner as infection estimation. Currently, this estimation is being performed semi-automatically (in a combined manual/automated scheme). A large shell is built that holds multiple copies of the program, one copy for each location being analyzed. Perturbation function code is inserted at the proper place in each location, and the shell is executed. Output from the shell is a set of $\mathrm{F}_{l,a}$ probabilities for each location $l$.

[4] Defined in Section 3.3.

[5] The failure propagation estimate is a point estimator which is the sample mean of the number of pseudo-failures, however to determine the confidence intervals in the estimate, the confidence interval of the frequency distribution for 95% confidence can be found with $p \pm w$, where $w = 2 \cdot \sqrt{(p \cdot (1-p)/n)}$ and $p =$ count $/n$ [4].

| *location* | *propagation estimate* |
|---|---|
| 1 | 0.764164000000 |
| 2 | 0.000000000000 |
| 3 | .4493680000000 |
| 4 | 0.18595600000 |
| 5 | 0.13760400000 |

Table 3.1: Propagation estimates for *qcksrt* using perturb(0.95,1.05,x) and perturb(0.5,1.5,x)

Empirical evidence exists to support a simplification: shorten the third step of the algorithm by perturbing on the first iteration data state only.[6] A program presented in Chapter 6 had failure propagation estimates determined in two manners:

1. as the algorithm specifies with perturbation only occurring in $f_l(\mathcal{B}_{l,1}(x))$, and
2. by repeatedly perturbing each data state $f_l(\mathcal{B}_{l,1}(x))$, $f_l(\mathcal{B}_{l,2}(x))$,...

It might be expected that as the perturbation parameters widened, failure propagation estimate values would increase. This did occur at two out of three locations with the above algorithm of determining failure propagation estimates as shown in Tables 6.4 and 6.5. However when the second scheme was applied, Table 3.1 shows that regardless of the perturbation function, identical estimates occurred. A more intuitive argument for perturbing on the first data state can be given by first answering a question: "when can a data state get infected?" Infection can occur on any iteration of the location, however there will be a first time that infection will occur. The earliest that infection can occur is on the first iteration, hence perturbation is performed on the first data state for this reason. On which iteration to perturb is a topic for further investigation.

### 3.2.2 Viral Propagation

The *viral propagation estimate* of location $l$, on the i$^{th}$ iteration of location $l$, $V_{l,i}$, is the probability that if a virus exists in $\mathcal{B}_{l,i}(x)$ for a randomly selected $x$, then there is a virus in $\mathcal{A}_{l,i}(x)$ for some $i$. Whether the error degree of the infected data state changes between data states is not considered (provided that the error degree

[6] Detailed in Chapter 6.

is greater than or equal to one after executing location $l$ on the i$^{th}$ iteration).[7] One algorithm for estimating $V_{l,i}$ at location $l$ is:

1. take a subset of data states that cause path $j$ to be reached; denote the set of inputs that created these data states as $\phi_j$,

2. let the subset of the input points represented by infected data states in $ds_{\mathfrak{P}_l}$ be denoted by the set $\{ds_{\mathfrak{P}_l}$ is infected$\}$.[8]

3. execute location $l$ on each data state represented by an input point in $\{ds_{\mathfrak{P}_l}$is infected$\}$,

4. then

$$V_{l,i} = \frac{\mid \{ds_{\mathfrak{S}_l} \text{ is infected}\} \mid}{\mid \{ds_{\mathfrak{P}_l} \text{ is infected}\} \mid} \tag{3.5}$$

which is the proportion of input points that managed to stay infected in the succeeding subdata space of location $l$ on the i$^{th}$ iteration of location $l$ on path $j$.

## 3.3 Perturbation Functions

*Potential faults* for a location are the set of faults that are considered to have a higher probability than some threshold of occurring. Since this set of faults will in general be unknown, in the ensuing discussion, consider the set of potential faults to be a set of "common faults." For instance, at a location that increments a counter, a potential fault might be to omit the location; inserting a compiler at the location is a ridiculous fault. An *infection interval* is a distribution of the range of values that the set of potential faults at a location may map a variable 's value into. Infection intervals are represented by the left distributions in each of graphs shown in Figure 3.2(in general the exact shape and orientation of this distribution will be unknown). The infection interval is a function of the preceding data space and the set of potential faults. Although the set of potential faults at a location may be infinite, limitations may be placed on this set by determining membership within the set according to impacts on values. The term infection interval is not related to the term infection rate; infection interval is introduced for convenience in the discussion in determining perturbation function parameters. *Perturbation*

[7] Typically a location will either increase or decrease the error degree, however for a location to change the virus would require it to both clean the current infection in $\mathfrak{B}_{l,i}(x)$ and create a new infection in $\mathcal{A}_{l,i}(x)$.

[8] Chapter 5 explains how to determine $\{ds_l$ is infected$\}$.

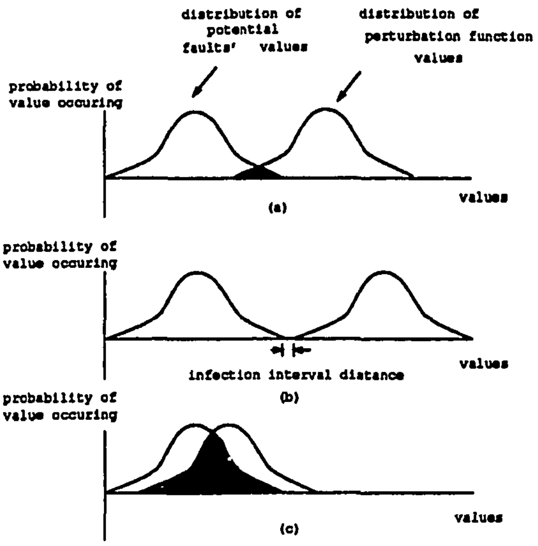

distribution of potential faults' values
distribution of perturbation function values
probability of value occuring
values
(a)
probability of value occuring
infection interval distance
values
(b)
probability of value occuring
values
(c)

Fig. 3.2: Example showing an infection interval versus potential fault value distribution

*function parameters* contain the information of the desired impact on a variable in a data state that is required in failure propagation estimation. The curves on the right of each graph in Figure 3.2 represent the distribution of values produced by particular perturba.ion parameters.

An *infection interval's length* is the difference between the largest and smallest value in the infection interval. The *infection interval's distance* is the linear distance along the abscissa-axis between the infection interval of potential faults and the distribution of values a perturbation function would map the same preceding data state into. If these ranges overlap the infection interval distance is zero. Figure 3.2(a) is an example where the distance is zero. Figure 3.2(b) is an example where the distance is not zero. Figure 3.2(c) shows an example where the two distributions are almost the same. A goal in finding perturbation function parameters is to cause as much overlapping between the curves in Figure 3.2 as possible. This assures impacts similar to those of potential faults.

Recall a perturbation function is the parameterized function that is used to infect data states during failure propagation analysis. Input to the function is the current value of the active variable being altered. Output is a new value in the range specified by the perturbation function parameters. Section 3.3 designs an algorithm for a perturbation function that produces altered values that mimic the impacts on values that the set of potential faults would produce, i.e., an infection interval distance of zero. Although the infection interval distance will in general be unknown, there will occasionally be circumstances under which a "better than random guess" as the perturbing distribution and perturbation function parameters is possible. A function that produces random values based upon some perturbing distribution is used during failure propagation analysis to avoid the possibility of estimating failure propagation from "programmer" unlikely faults. This is a major complaint against fault-seeding techniques.

As an example of a "better than random guess", a constant function such as the following might be used during failure propagation analysis if it were known that any fault placed at the location under analysis would produce one particular value $z$ in the succeeding data state:

$$f(x) = \left\{ z \quad \text{for all x} \right.$$

Such a perturbation function would be practical if it were known that a programmer was misinformed that a variable should be set to constant when it should be set to constant + 1; such will be a rare event at best. This can be generalized to a set of constants as well with the function

$$f(x) = \left\{ z \quad \text{for all x, where } z \in \text{a set of constants} \right.$$

This might occur in a compiler in the parse tables.

If the perturbation function were to produce the same output value as the input value, no failure propagation is possible leaving the failure propagation estimate biased. If the perturbation function always produced the original value the failure propagation estimate would be zero. In general, the type of function used will produce random values based upon the input value, a perturbing distribution, and the perturbation function parameters. The *perturbing distribution* will in general be a random distribution, and the *perturbation function parameters* are the parameters to the perturbing distribution.

A twofold concern exists in determining a non-constant perturbation function: first is determining the perturbing distribution; second is determining the parameters this function. The remainder of this section addresses a scheme for the latter of these concerns under the assumption that the perturbing distribution is uniform. This scheme can be generalized to a non-uniform two parameter perturbing distribution.

Possibilities for non-constant perturbing distributions include all of the continuous and discrete random distributions. The perturbation function newvalue(x) := equilikely( trunc(oldvalue(x)*0.6), trunc(old- value(x)*1.40)) is an example of a discrete distribution that perturbs a value by substituting an equilikely random value on the interval of 40% more and 40% less than the original value. For this function, the infection interval length is trunc(oldvalue(x)*1.4) - trunc(oldvalue(x)*0.6) = trunc(oldvalue(x)*0.8). This function however leaves the possibility of returning newvalue(x) = oldvalue(x). The following code avoids this. Note two things about this fragment: if the value being perturbed is zero, either a -random_park or random_park is returned, and if constant1 and constant2 are so close to 1.0 that the while loop does not normally exit, the loop is forced to halt after five iterations and either increase or decrease the input value by random_park. Note that the call to random_park is a call to the random number generator in [2] which generates a real value between 0.0 and 1.0.

```
function perturb(x: real;): real;
var        newvalue: real;
constant1: real;
constant2: real;
counter  : integer;
begin
constant1 := (*0.0 < constant1 < 1.0*)
constant2 := (*constant2 > 1.0*)
newvalue := x;
if (x=0.0) then begin
           if random_park < 0.5 then
            newvalue:=random_park
           else
            newvalue:=-random_park
          end
else
begin
          counter := 0
          while (newvalue = x) do begin
             newvalue:= uniform(x*constant1, x*constant2);
             counter := counter + 1;
             if (newvalue=x) and (counter=5) then begin
                if random_park < 0.5 then
                  newvalue:=x-random_park
                else
                  newvalue:=x+random_park;
             end;
```

```
        end;
end;
perturb  := newvalue;
end;
```

The choice of a perturbation function is an important decision; one should have an idea of the length and location along the abscissa-axis of the infection interval before determining the perturbation function parameters. If the impacts of the potential faults are expected to make modest changes to the value, and each of these changes is likely, then equilikely( trunc(oldvalue(x)*0.9), trunc(oldvalue(x)*1.1)) or equilikely( trunc(oldvalue(x)*0.95), trunc(oldvalue(x)*1.05)) are reasonable choices since they have small infection interval lengths. The motivation in determining perturbation function parameters is that there is an assumption that a fault that produces a distribution of values that is linearly far from the infection interval along the abscissa-axis should be caught during developmental testing. This means that the greater the divergence in values from faults, the decrease in the chance of type I cancellation. If a fault maps values outside the range of the infection interval, the observed propagation rate should be greater than the propagation estimate. This notion requires additional research. If the analysis is performed with parameters that produce a smaller range of values, then a more conservative estimate of propagation results. Even if the fault causes a smaller range of values than anticipated, there is uniform sampling from the points which occur in the smaller interval.

A method of determining the perturbation function parameters for a two parameter perturbing distribution where the parameters define the upper and lower bound for the outputted value is:

1. make the assumption the infection interval has endpoint parameters $a, b$, thus $[a, b]$ is the interval,

2. decrease this interval to $[a + c, b - c]$, where $c > 0$, $c \ll b - a$, and the value for $c$ is arbitrary,

3. pick the value $a + c$ or a value near $a + c$ and check it's propagation; if it propagates, reset $a$ to this value, and try the value $b - c$ or a value near $b - c$; if it propagates, reset $b$ to this value, and try a value slightly greater than $a + c$; if it propagates ignore it and try a value slightly less than $b - c$; repeat many times in this step-wise manner until one of the following is true:

   (a) all values tried in $[a,a + c]$ and $[b,b - c]$ propagated, in which case the parameters $a, b$ are chosen for the perturbation function,

(b) the first point $s \in [a, a+c]$ did not propagate, hence the parameters are $s, (b-(s-a))$, or

(c) the first point $s \in [b, b+c]$ did not propagate, hence the parameters are $(a+(b-s)), s$.

It is unlikely that the set of potential faults will produce altered values that can be correlated with any distribution as implied in Figure 3.2. There could be many spikes along the abscissa-axis appearing as random "noise." The algorithm presented above can be adjusted if the potential faults distribution is known. However in general this is not known, therefore the uniform distribution is assumed. Another distribution worth noting for potential use as a perturbing distribution is the normal distribution, where the mean of the distribution is set at the original value. The normal distribution would be a reasonable perturbing distribution if the potential faults at the location are more likely to make small changes as opposed to large changes. The results of this thesis are found in Chapter 6 and used the uniform perturbing distribution. Additional studies on producing perturbation functions that mimic potential faults is an area for future research.

## 3.4 Conclusions

Infection rate, propagation rate, and execution rate are defined for a known fault at a specific location. The definitions from Chapter 2 are for known faults and the definitions provide a basis for understanding how specific faults are related to the failure rate. When testing begins, the faults are naturally not known, and propagation rates and infection rates can not be directly measured. So Chapter 3 provides both a model and implementation for generalizing the definitions of infection rate and propagation rate to produce the infection estimate and propagation estimate. This generalization is performed by hypothesizing that a fault exists at a location and measuring the impact of the hypothesized fault [3]. Infection estimation performs this with semantic alternatives; viral propagation estimation performs this with semantic alternatives as well; and failure propagation estimation performs this generalization with perturbation functions. Research is continuing into designing perturbation functions, specifically how various perturbing distributions impact failure propagation analysis.

Chapter 3 has defined the model and implementation for the PIA methodology. Chapter 4 presents additional algorithms necessary for implementing the model.

# References

[1] R. DeMillo and D. Guindi and W. McCraken and A. Offutt and K. King. An extended overview of the mothra software testing environment. *Proceedings of Second Workshop on Software Testing Verification and Analysis*, July 1988.

[2] Stephen K. Park and Keith W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, October 1988.

[3] Jeffrey M. Voas and Larry J. Morell. *Fault Sensitivity Analysis(PIA) Applied to Computer Programs.* Technical Report WM-89-4, College of William and Mary, Department of Computer Science, December 1989.

[4] Steve Park. *Lecture notes on simulation, Unpublished.* College of William and Mary, August 1988.

[5] ByoungJu Choi and Aditya P. Mathur and Brian Pattison. P$^M$othra: scheduling mutants for execution on a hypercube. *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, pp. 58–65, December 1989.

[6] Larry J. Morell and Jeffrey M. Voas. *Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability.* Technical Report WM-88-2, College of William and Mary, Department of Computer Science, September 1988.

[7] William Mendenhall and Richard L. Scheaffer and Dennis D. Wackerly. *Mathematical Statistics with Applications.* Duxbury Press, 1981.

# Chapter 4

# Propagation and Infection Methodology

This chapter describes modifications to the algorithms of Chapter 3 for gaining efficiency and improved quality in the estimates. Figure 4.1[1] shows the interaction between the main and auxiliary processes of the implementation discussed in this chapter.

Propagation and infection analysis may eventually be extended into a general purpose scheme for analyzing arbitrary programs, however the application of propagation and infection analysis has only been to programs written in Pascal and Fortran-77; thus claims about propagation and infection analysis's applicability to arbitrary languages are not made. PIA requirements on the input program are:

1. the program is written in a structured language [10],[2]

2. the program is closely correct; a program $P$ is termed to be *closely correct* if and only if

   (a) $P$ compiles,

   (b) testing schemes currently being applied are producing no failures (therefore it is assumed testing is to be halted),

[1] In Figure 4.1, $\boxed{\text{process 1}} \xrightarrow{a} \boxed{\text{process 2}}$ represents information $a$ which is created in process 1 and passed to process 2. $P$ is the original input program.

[2] The Fortran-77 software considered in Chapter 6 is from the LIC experiment[14] and this software does not satisfy this criterion.

(c) $[P] \approx [P_c]$ where $P_c$ is a correct version of $P$; $\approx$ in this case may be quantified to a proportion of inputs of less than $10^{-4}$ on which these two functions differ, and

(d) the failure rate of $P$, $\lambda_P$, is $< 10^{-4}$ failures per execution.,[3]

3. all program variables are of a statically declared size.

The first requirement exists in order to limit the types of paths through a program; this simplifies the path equivalence class definitions and the program flowgraph. The second requirement is needed because propagation and infection analysis is temporally expensive. The intuition behind requiring the closely correct criterion is to ensure that the program structure changes minimally as faults are removed; the four requirements are a stringent attempt to assure this. This means that propagati and infection analysis will not need to be reperformed which would be temporally expensive. Requirement 2(c) ensures that for most inputs $x$, $[P](x) = [P_c](x)$. Requirement 2(d) ensures that those inputs $y$ where $[P](y) \neq [P_c](y)$ have low probabilities of being picked. The third requirement exists to simplify sampling and storage of data states. It is not a theoretical limitation of propagation and infection analysis, it is a practical limitation; it exists to simplify the implementation of the process that produces data states. There does exists a propagation and infection analysis implementation scheme that can handle dynamic variables in the input program which will be explained in this chapter.

The use of the term "program" when discussing the application of propagation and infection analysis can also be interpreted as "module." A module has an input distribution (from values of global variables and in parameters) and the global variables defined in the module combined with the out parameters are the module's output. For some large software systems, there are modules whose integrity are far more critical than others. For such modules as well any other module, propagation and infection analysis may be individually applied.

The processes that this chapter discusses and are represented in Figure 4.1 are:

- Process Simplify
- Process Abstraction Analyzer
- Process Dataflow Analyzer

[3] $\lambda_P < 10^{-4}$ failures per execution is what was used for the experiments of Chapter 6. This value was determined arbitrarily.

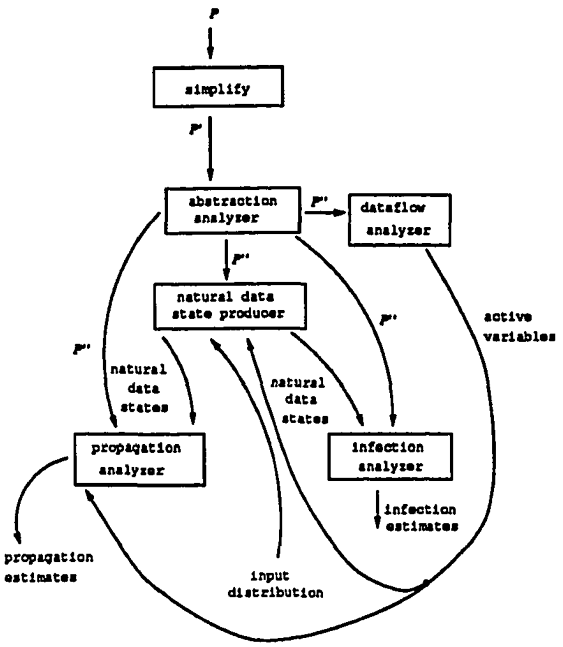


Fig. 4.1: Processes of PIA

- Process Natural Data State Producer
- Process Propagation Analyzer
- Process Infection Analyzer

Sections 4.1, 4.2, 4.3, and 4.4 describe auxiliary processes that convert the input program and input distributions in a manner such that the main propagation and infection processes are more specific and more efficient. Sections 4.5 and 4.6 explain the propagation estimation process and infection estimation process. Section 4.7 describes the final output produced by the main processes. Section 4.8 explains how the final output described in Section 4.7 relates to the objective of the thesis set forth in Chapter 1.

## 4.1 Process Simplify

Process Simplify is an auxiliary process and is the first process of the methodology invoked. Process Simplify inputs the original program $P$ and produces a program $P'$ such that $[P'] = [P]$. This process performs two tasks. The first task takes arithmetic expressions in assignment statements of $P$ with $k$ operators($k \geq 2$) termed *compound expressions* and produces functionally equivalent arithmetic expressions in $P'$ with $k$ expressions each with one operator. An arithmetic expression with one operator and two operands is termed a *simple expression* and is in the form $a \odot b$, where $\odot \in \{+, -, *, /\}$ and $a$ and $b$ are variables or constants. This task requires the introduction of additional local variables. The reduction to simple expressions follows the precedence of the original expression, so if $P$ contains a precedence fault, so will $P'$.[4] The BNF grammar for arithmetic expressions whether simple or compound [13] is:

```
<arithmetic expression> ::= <term> | <arithmetic expression> + <term>
                          | <arithmetic expression> - <term>
<term>       ::= <factor> | <term> * <factor> | <term> / <factor>
<factor>     ::= <variable> | <constant> | (<arithmetic expression>)
```

The second task inputs *predicates* from $P$ which are boolean combinations of arithmetic relational expressions. The logical operators and and or form boolean combinations. Each arithmetic relational expression contains a relational operator from $\{<, >, =, \neq, \leq, \geq\}$. Predicates in the form if (a) and (b) then and if (a) or (b) then are termed *compound predicates* [1]. Process Simplify converts compound predicates of this form to if (a) then begin if (b) then .... end and if (a) then ... else if (b) then .... respectively. These (a)s and (b)s are termed *simple predicates* [1] when they are in the form $x \odot y$, where $\odot \in \{<, >, =, \neq, \leq, \geq\}$ and $x, y$ are arithmetic expressions. Arithmetic expressions in simple predicates are currently not simplified in the methodology. Potentially it may be shown that simplifying arithmetic expressions within predicates handles a particular situation of which the author is currently unaware. An example of what this entails follows:

```
if ((a+b+c) > d) then
```

[4] This problem is partially solved by requiring that variables have associated units; the Integrated Verification and Testing System (IVTS)[2] is an environment requiring units for programs in the language Hal/S[5, 9].

becomes:

```
dd = a + b
e = dd + c
if ((e) > d) then
```

Note that two additional local variables where created during simplification: dd and e. The BNF grammar for compound and simple predicates is:

```
<simple predicate>   ::= (<arithmetic expression> <relop>
                         <arithmetic expression> )
                         | not (<arithmetic expression> <relop>
                         <arithmetic expression> )
<compound predicate> ::= (<simple predicate> <andor> <simple predicate>) |
                         not (<simple predicate> <andor> <simple predicate>)
<andor>              ::= and | or
<relop>              ::= = | ≠ | <| > | ≤ | ≥
```

Implementation of Process Simplify is not required in PIA. Conversion to simple expressions and simple predicates encourages isolation of locations where faults of low impact can more easily exist without being found. Remember that propagation and infection analysis is structure-oriented; the more the software structure is decomposed, the more precise the analysis. Process Simplify is one tool designed specifically for achieving this. By implementing Process Simplify, a perturbation function has fewer fault impacts to simulate than when applied to compound expressions.

### 4.1.1 Simple Expression Examples

This section presents two examples of the conversion from a compound expression to a simple expression. Observations during experimentation have shown that performing simplification increases the preciseness of the quantification of the failure propagation estimates strictly through increasing the number of estimates. An example demonstrating this is the following compound expression from [19]:

```
t := 0.9 * (sqr(y+1.0)+1.0)*exp(em*glalxm-gammln(em+1.0)-glg)
(**** data space ****).
```

When failure propagation analysis is performed on the variable t of this expression in the succeeding data space, it produced a high failure propagation estimate. When failure propagation analysis is applied to the local variables being defined for the equivalent simple expressions:

```
aa := sqr(y+1.0);
(**** intermediate data space ****)
bb := aa + 1.0;
(**** intermediate data space ****)
cc := em + 1.0;
(**** intermediate data space ****)
dd := gammln(cc);
(**** intermediate data space ****)
ee := em*glalxm-dd-glg;
(**** intermediate data space ****)
ff := bb * exp(ee);
(**** intermediate data space ****)
t := 0.9 * ff;
(**** data space ****)
```

it is discovered that one of the simple expressions produces a tiny failure propagation estimate(close to 0.00003). This small estimate was not discovered when the compound expression was analyzed for failure propagation.

Creation of more locations by Process Simplification adds additional data spaces in addition to new variables. Data spaces in $P'$ but not in $P$ are termed *intermediate data spaces.* Intermediate data spaces represent the state of the computation after the newly created locations are executed. Intermediate data spaces contain the state of the computation at the subexpression abstraction level. This example shows that the higher the granularity(meaning the lower the abstraction level analyzed), the more precise the estimates produced.

One last example of expression simplification is the arithmetic expression x := (a*b)*ln(23 / k) which has 4 operators. The simple expressions are:[5]

```
v1 := a*b
(**** intermediate data space ****)
```

[5] The effects of the unary ln operator are ignored; there is no need to assign a unary operator and its parameter to a newly created variable; all that is required is that the argument to a unary operator is a simple expression or variable.

```
v2 := ln(23 / k)
(**** intermediate data space ****)
v3 := v1 * v2
(**** data space ****)
```

With intermediate data spaces, failure propagation analysis will simulate fault impacts in the succeeding data states of a*b, ln(23 / k), and v1 * v2 with three functions, instead of only simulating the fault impacts in (a*b)*ln(23 / k) by one perturbation function.

### 4.1.2 Simple Predicate Examples

Since infection analysis of conditional locations needs semantic alternatives which impact the program counter, conversion to simple predicates helps ensure that complete infection analysis occurs particularly when infection analysis is performed manually. As the number of simple predicates in a compound predicate increases, the probability of forgetting to analyze a semantic alternative increases when done manually and without simplification. By creating more locations which require less analysis, this probability is hopefully reduced. If infection analysis is automated, simplification of conditional predicates is unnecessary.

When predicate simplification occurs, the algorithm for failure propagation in Chapter 3 changes slightly. Failure propagation analysis at predicate locations should be performed in the data space of the last simple predicate that replaced a compound predicate. Do not perform failure propagation analysis in intermediate data spaces created by predicate simplification.[6] The reason for predicate simplification is improved infection analysis, not failure propagation analysis.

An example of conditional predicate simplification follows for both the if-then-else statement and the while-do statement. The compound predicate:

```
if (((a+b = c) and (d+e <= f) and (g=h)) or (a=d)) then
(**** data space ****)
  xxx;
```

is simplified to:

[6] Perturbing at an intermediate data space created by predicate simplification is equivalent to forcing a predicate to have side-effects which do not impact the program counter; this is unrealistic and does not occur; thus it is not performed.

```
if (a+b = c)then begin
(**** intermediate data space ****)
             if (d+e <= f) then begin
(**** intermediate data space ****)
                 if (g=h) then
(**** data space ****)
                     xxx;
                end;
end
else
if (a=d) then
(**** data space ****)
               xxx;
```

Notice that within the predicate a+b = c there is no simplification applied to the expression a+b. Predicate simplification only applies to the logical operators, however to increase the granularity of the analysis this may be advisable.

As an example of the while-do statement, consider the following:

```
while (((a+b = c) and (d+e <= f) and (g=h))) do
(**** data space ****)
  xxx;
```

This is simplified to:[7]

```
100: if (a+b = c)then begin
(**** intermediate data space ****)
                  if (d+e <= f) then begin
(**** intermediate data space ****)
                      if (g=h) then
(**** data space ****)
                          xxx;
                          goto 100;
```

[7] With the use of a goto; the usage of a goto is allowed because its inclusion does not alter the overall flowgraph structure. Note that the goto location created does not receive failure propagation analysis or infection analysis since it clearly is going to the correct location. If the goto transfers control to the wrong location on a particular iteration, that is because the predicate is wrong.

```
end;
```

There is an implicit assumption in the implementation of Process Simplify that [original software] = [simplified software], i.e., the simplification process outputs functionally equivalent software. It should be clear that all the transformations described in this section are semantic-preserving.

## 4.2 Process Abstraction Analyzer

Process Abstraction Analyzer is a necessary auxiliary process that inputs $P'$ and outputs a program $P''$ with each level 1 location identified by a descriptor containing a unique location number. This process is trivial and can be performed manually or automatically.

## 4.3 Process Dataflow Analyzer

Process Dataflow Analyzer is a necessary auxiliary process that inputs $P''$ and outputs the set of active variables at each location regardless of the succeeding subpath. Hence, Process Dataflow Analyzer produces the active variables at each data space.[8] For each data space in $P''$, Process Dataflow Analyzer internally produces the program dependence directed graph. The active variables are then determined from the reflexive transitive closure of the program dependence directed graph. This procedure is static. References to dataflow analysis include [7, 8, 6, 3].

## 4.4 Process Natural Data State Producer

Process Natural Data State Producer is a necessary auxiliary process for PIA. Infection and propagation estimates are functions of sampled data states; Process Natural Data State Producer is responsible for providing these sampled data states. Process Natural Data State Producer inputs $P''$ and the input distribution of $P$, and outputs the value distributions for each data space and intermediate data space of $P''$. Process Natural Data State Producer exists to ensure that

[8] This process can determine the active variables at subdata spaces if augmented with the information about the path equivalence classes.

the sampled data states used during failure propagation and infection analysis resemble those that the program produces during execution.

The process of forming sampled data states should ensure that there exists an input point for the program that could have created that data state. For instance, if the predicate ($x_1 > 100 \iff x_2 > 500$) is true at every data space in the program regardless of input, then the sampled data state ($x_1 = 200$, $x_2 = 20$) should not be created. A sampled data state that is not producible from an input point is termed an *artificial data state*; sampled data states meeting the requirement are termed *natural data states.*

### 4.4.1 Trade-offs Between Artificial and Natural Sampled Data States

How important is it that sampled data states are not artificial? Can failure propagation or infection estimation be influenced by a particular sampled data state? Intuitively, an artificial data state could bias the failure propagation estimate since a program could exhibit more or fewer pseudo-failures from internal data states on which it was never intended to execute. An artificial data state could also bias an infection estimate for a particular semantic alternative. Artificial data states may adversely affect control flow during failure propagation analysis. This leads to Hypothesis 4.1:

**Hypothesis 4.1** *Using natural data states as sampled data states rather than artificial data states may decrease the chance of biasing the estimating process.*

The argument for Hypothesis 4.1 follows:

**Comment on Hypothesis 4.1** This comment on how artificial data states may bias analysis results is in two sections: one for how artificial data states may affect failure propagation analysis and one for how they may affect infection analysis. (I.) For a section of code $f$ whose type I cancellation is being analyzed, if for every data state $x$ regardless of whether artificial or natural, $[f](x) \uparrow$ and $f$ is a one-to-one mapping from its input domain to its output domain, then it does not matter during failure propagation analysis whether the data states are natural or artificial. However consider the one-dimensional input space for the subprogram $f$ where

$$[f](x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

and where all the natural data states prior to $f$ correspond an input value greater than zero, and thus the one-dimensional artificial data states correspond to those values less than or equal to zero. If artificial data states are used in determining the failure propagation estimates of $f$, then the estimates will be a function of $[f](x) = 0$ instead of $[f](x) = 1$. This certainly could bias the failure propagation estimate if different locations are involved in computing $[f](x) = 0$ versus computing $[f](x) = 1$. (II.) For infection analysis, suppose that at location $l$,

$$[l](x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

and for a particular semantic alternative $a$,

$$[a](x) = \begin{cases} 1 & \text{if } x > 10^6 \\ 0 & \text{otherwise.} \end{cases}$$

Now suppose that for the one-dimensional data state space prior to location $l$, the natural data states include those values less than zero and greater than $10^6$. If the artificial data states are used which are in the range $[0...10^6]$, then infection estimates will be greater than zero when they should be zero.

Hypothesis 4.1 is a topic for future research. Natural data states are preferable since they are created from either known executions or some "higher" knowledge of either the program or the input distribution; propagation and infection analysis attempts to make an "expected" statement about "what might occur tomorrow" from "what has occurred today". Using as much information known "today" about "what is expected to occur tomorrow" can only make the "what might occur tomorrow" statement better. Natural data states have the advantage in doing this. So the concern that exists in producing natural data states is ensuring that the values chosen in the sampled data state of a location are producible from an input point that reaches that location.

The algorithms presented in Section 4.4 for producing natural data states address this concern. A simple example demonstrating the need for natural data states is: `if (a> 0) then b := b / a`. If no regard is made to the values `a` would normally have before the assignment statement, and values for `a` are randomly chosen, potentially `a` will be assigned a sampled value of zero. Run-time termination during Process Propagation Analyzer or Process Infection Analyzer will then occur forcing the process to be aborted. This may cause previous analysis prior to termination to be lost. A second example advocating Hypothesis 4.1 is more direct: suppose that a function is used which generates artificial data states at

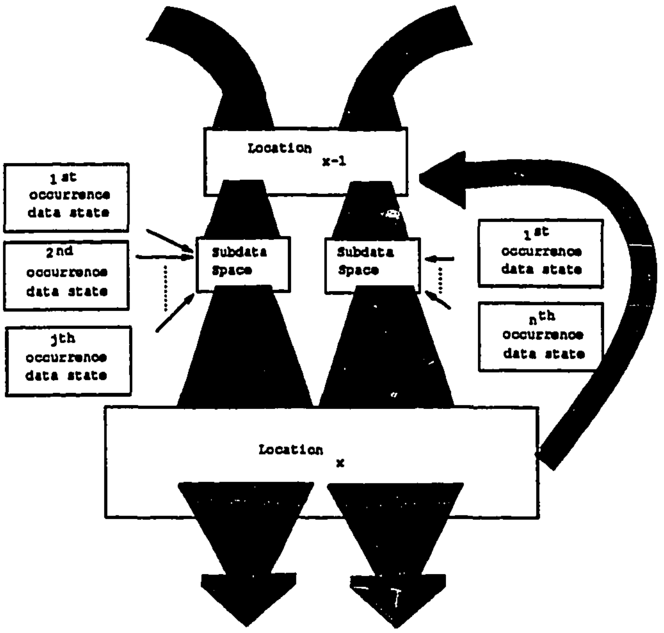

Location x-1
1st occurrence data state
2nd occurrence data state
jth occurrence data state
Subdata Space
Subdata Space
1st occurrence data state
nth occurrence data state
Location x

Fig. 4.2: Data space value distributions for repeated locations

some location. And suppose that these artificial data states frequently cause an infrequently executed branch to become a frequently executed branch during failure propagation analysis. Further suppose that this branch which is frequently traversed contains locations which cause frequent type I cancellation. Then failure propagation of the location under consideration will be underestimated.

A "mini" experiment was performed using a quadratic program which explores this question further.[9] The program used for the experiment is Quadratic.[10] The results are as follows: when natural data states were used, failure propagation estimates of 1.0 occurred 90% of the time, and failure propagation estimates between 0.6 and 0.7 occurred for the remaining 10% percent; however when artificial data states were used, failure propagation estimates were always 1.0. So in this "mini" experiment, artificial data states overestimated the failure propagation estimates made with natural data states.

[9] The experiment was performed on only one program due to the intuition that artificial data states bias estimates; this idea is purely intuitive, and by no means validated.

[10] Found in Chapter 6.

### 4.4.2 Creation of Value Distributions at Data Spaces

This section describes the temporal and spatial requirements of three schemes of producing natural data states. Recall that a value distribution is a structure for storing data states at a data space. Figure 4.2 portrays the flow between two successive locations within a loop where $\forall i\ \exists y\ succ(x-1,i,y) = x$ . Figure 4.2 shows two path equivalence classes coming into these two locations, and the corresponding subdata spaces created by each path between two locations. The occurrence data state boxes in Figure 4.2 on the left for the first through j$^{th}$ iteration represent the sequence $[\mathcal{A}_{x-1,1}(a), \mathcal{A}_{x-1,2}(a), ..., \mathcal{A}_{x-1,j}(a)]$ for some input $a$. The occurrence boxes in Figure 4.2 on the right for the first through n$^{th}$ occurrence represent the sequence $[\mathcal{A}_{x-1,1}(b), \mathcal{A}_{x-1,2}(b), ..., \mathcal{A}_{x-1,n}(b)]$ of data states for some input $b$. Of the three methods for producing natural data states to be presented in Section 4.2, only one method produces value distributions. The second method produces symbolic expressions which can be used to generate natural data states and the third method creates natural data states from executing predecessor locations on an input point [4]. The implementation of the method chosen for creating natural data states depends upon the abstraction level the analyzing process requires. For example, the value distribution for a variable $z$ at location $k$ is effectively the result of overlaying the value distributions at each subdata space of location $k$ that represents a path equivalence class on which variable $z$ is active at location $k$. Value distribution creation for data spaces requires less computation than for subdata spaces since it is not necessary to determine the proper subdata space that a data state belongs included in. The main difference between implementing value distribution creation at data spaces versus subdata spaces is that until determination is made as to which path equivalence class the data states correspond, all data states executed must be stored temporarily. Once the determination is complete, the appropriate subdata space value distribution is updated.

In Sections 4.4.2.1-4.4.2.3, three different schemes for creating natural data states are presented; these are referred to as Method I, Method II, and Method III; these are three implementations for Process Natural Data State Producer. Method I produces natural data states for a location by storing a symbolic expression for each active variable of each data space in terms of all computations of that variable prior to that location. This requires a set of expressions for each preceding subpath to the location. The set of expressions combined with the corresponding elements of the input domain, $\phi$, for that subpath suffice in satisfying the natural data state criterion; these expressions are effectively symbolic execution expressions. Method II executes the program a large number of times and builds value distributions containing the data states created during execution. Method III requires no storage and effectively never builds value distributions. It simulates both the creation of value distributions and sampling from them by being a "subprocess" that is inserted into the main PIA processes which require sampled data states. In effect, Method III generates natural data states as it needs them by executing all preceding locations to the location where natural data states are needed. The disadvantage of Method III is that a main process must re-execute enormous quantities of code.[11]

The following discussion of these three schemes is presented for producing natural data states at the higher abstraction level which is the subdata space. This scheme is presented since this implementation is more complicated than an implementation at the data space abstraction level. Conversion to an equivalent algorithm for determining natural data states of a data space involves ignoring all references in the ensuing discussions to waiting until path determination is made.

#### 4.4.2.1 The Algorithm for Method I

The algorithm for Method I is presented through example using the following code segment:

```
read(a) [1]
(**** data space 1 ****)
read(z) [2]
(**** data space 2 ****)
if a > 0 then [3]
(**** data space 3 ****)
  a := a - z [4]
  (**** data space 4 ****)
else
  a := a + z [5]
  (**** data space 5 ****)
a := sqr(a) / z
(**** data space 6 ****)
```

Assume at $ds_{g_6}$ variables a and z are active. The algorithm for Method I consists of storing two types of information: an expression for each active variables at each subdata space, and either expressions for what the initial input values are that cause a particular path equivalence class to be executed at each subdata space,

[11] The use of a multi-processor machine can help with this problem. Any speed up will be a function of the number of processors.

| data space<br>or subdata space | active variable in<br>in expression form | input condition |
|---|---|---|
| 1 | a = input | |
| 2 | a = input<br>z = input | |
| 3 | a = input<br>z = input | |
| 4 | a = a - z<br>z = input | |
| 5 | a = a + z<br>z = input | |
| 6 | a = $(a+z)^2/z$<br>z = input | $a \leq 0$ |
| | a = $(a-z)^2/z$<br>z = input | $a > 0$ |

Table 4.1: Method I example

or the actual input points of a particular path equivalence class at each subdata space. Initially, $\phi$ is divided into equivalence classes according to $PEC$; symbolic expressions are then constructed containing information about the computation that occurred to an active variable on a preceding subpath. For this example, at $ds_{g_6}$, two expressions occur for a at $ds_{g_6}$: (a + z)$^2$ / z and (a − z)$^2$ / z. At $ds_{g_6}$ there exists two subsets of $\phi$: one containing values when $a > 0$; the other containing values when $a \leq 0$. With the partitioning of $\phi$ and the arithmetic symbolic expressions, values for a and z can be generated at $ds_{g_6}$ that satisfy the natural criterion. For this example, Table 4.1 shows the information stored by Method I.[12]

Partitioning $\phi$ according to the path equivalence classes may be partially determined by executing $P''$ many times. Putting every element of $\phi$ in its proper partition would require executing $P''$ on every element in $\phi$ which may be intractable. Determination of the path equivalence class for a particular input point is performed as follows: execute $P''$ with proper instrumentation to tell when a location is reached; then pattern match the output string against the known regular expressions of $PEC$ to determine the exact path equivalence class. Then add the input point to the correct partition of $\phi$.

Method I can be automated to generate the symbolic expressions provided it has three parameters: $\phi$, the path equivalence classes, and $P''$. By backtracking through previous expressions and substituting these expressions into references

[12] "a = input" in Table 4.1 and Table 4.2 denotes that variable a has not been changed since being inputed.

| data space<br>or subdata space | active variable in<br>in expression form | input condition |
|---|---|---|
| 1 | a = input, pc = 2 | $\forall a$ |
| 2 | a = input, pc = 4 | $a \geq 5$[13] |
|  | a = input, a = input + 1, a = input + 2 ,...<br>$pc = \begin{cases} 3 & a < 5 \\ 4 & a \geq 5 \end{cases}$ | a <5 [14] |
| 3 | a = input + 1, a = input + 2,... pc = 2 | a < 5 |

Table 4.2: Method I loop example

in the current expression, previous computations affecting a variable are stored. Loops in the input program make Method I more difficult, since at each location within a loop a separate expression is needed for each potential previous execution. In order to make this feasible, there must be a method for collapsing the potentially infinite number of expressions a loop creates. In the case of loops, it will be necessary to have an identification for the expression representing the expression for the first time at the subdata space. Table 4.2 shows the information necessary for the following indefinite loop.

```
read(a) [1]
(**** data space 1 ****)
while (a<5) do [2]
(**** data space 2 ****)
  a := a + 1 [3]
  (**** data space 3 ****)
. . . . . [4]
```

A mean-value analysis of the spatial requirements for Method I is now given; the temporal requirements are ignored to generate the expressions and to generate the partitions of $\phi$. The assumption is made that the space required to identify expressions for the first time through a location is negligible. Let $t = \mid T \mid + \mid TS \mid$, $\mu_t$ = the mean number of locations per path equivalence class, $\mu_j$ = the mean number of times through a loop, $\mu_r$ = the mean number of locations which may

[13] This is for the subdata space corresponding to the path equivalence class for never executing the loop body.

[14] This is for the subdata space corresponding to the path equivalence class for executing the loop body at least once.

be repeated on a path equivalence class, and $\mu_a$ = the mean number of active variables per location.

Then $\mu_t - \mu_r$ is the mean number of locations on a path equivalence class which are never repeated. For these non-repeated locations, $(\mu_t - \mu_r)\mu_a$ represents the number of expressions needed for one location for one path equivalence class; then for all path equivalence classes just multiply this expressions by $t$ giving: $t(\mu_t - \mu_r)\mu_a$ expressions needed for all path equivalence classes for all non-repeated locations. Similarly, for locations that are repeated, the mean-value of the number of required expressions is $t\mu_r\mu_a\mu_j$.[16] So the mean number of expressions required in storage for creating natural data states is $\Theta(t(\mu_t - \mu_r)\mu_a + t\mu_r\mu_a\mu_j)$. If the mean amount of space required to store one expression is $s$, then the space order of Method I is $\Theta(st(\mu_t - \mu_r)\mu_a + st\mu_r\mu_a\mu_j)$.[17] As a rough estimate, if the following assignments are made: $t = 500$, $\mu_t = 100$, $\mu_j = 10$, $\mu_r = 10$, $\mu_a = 50$, and $s =$ 1 kilo-byte, this example requires 4.864 giga-bytes. If the amount of computational time required to solve one expression is symbolically represented by $\lambda$, then the mean amount of time required to get a natural data state is $\Theta(\mu_a\lambda)$.[18] In summation, Method I is a high-overhead scheme which is not advised.

#### 4.4.2.2 The Algorithm for Method II

Method II is a dynamic scheme that requires executing $P''$ many times. It is performed by augmenting $P''$ with both a procedure to announce that a location has been reached and a procedure to store data states observed during an execution. Method II is the only one of the three methods that actually builds value distributions.

Before execution of the Method II implementation of Process Natural Data State Producer can begin, $P''$ is instrumented with "write" statements to print out a message as each location has been reached. This produces a long message(string) that is matched against the list of regular expressions representing the path equivalence classes of $P''$. In addition to the "write" statements, $P''$ is augmented with an algorithm to temporarily store the data states from the execution until pattern matching is complete; then the data state is entered into the value distribution of the correct subdata space.

The algorithm for the Method II implementation is described through the

[16] Note that $\mu_j$ accounts for cases such as nesting of loops within loops since it represents the average number of times a repeated location is repeated.

[17] The space needed to store the equivalence classes of $\phi$ is considered negligible.

[18] $\lambda$ includes the time to obtain element of $\phi$ and the time to solve the expression.

following example. Let $u$ be the number of statically declared variables in $P''$. Let $z$ be the maximum number of active variables at any location in $P''$ , and let $y$ be the maximum number of active variables for any path equivalence class of $P''$.[19] So then $z \leq y \leq u$. The output from Process Dataflow Analyzer and Process Abstraction Analyzer[20] contains enough information to determine the variables in the subdata spaces. Now suppose $P''$ has three path equivalence classes, $i$, $j$, and $k$, and that each path equivalence class has location $x$ represented in its regular expression. Let $\mathsf{A}_i = \{a_{i,1}, a_{i,2}, a_{i,3}, \ldots\}$ represent the set of active variables at data space $x$ on path equivalence class $i$, let $\mathsf{A}_j = \{a_{j,1}, a_{j,2}, a_{j,3}, \ldots\}$ represent the set of active variables at data space $x$ on path equivalence class $j$, and let $\mathsf{A}_k = \{a_{k,1}, a_{k,2}, a_{k,3}, \ldots\}$ represent the set of active variables at data space $x$ on path equivalence class $k$. And let $\mathsf{A}_i$, $\mathsf{A}_j$, and $\mathsf{A}_k$ represent all subdata spaces at data space $x$. Then $\mathsf{A}_i \cup \mathsf{A}_j \cup \mathsf{A}_k$ is the set of all active variables at data space $x$.

Execution of $P''$ reaches a given location and yields a natural data state for that location for a particular path equivalence class. At this point, the identification of the path equivalence class to which the data states belongs has not occurred, so the data state may not be included into the value distribution for any subdata space. What is known is that there are $u$ variables for the program. So for each data space along the trip there is an $u$-tuple of values, where any uninitialized or dead variable may be considered zero, or assigned a value representing a dead status. For each naturally occurring data state occurring during execution, Method II stores one $u$-tuple of values of a node in a dynamically allocated linked list. This produces a list of natural data states for each execution. The size of a list node must be large enough to hold $u$ values and an identification number of the data space it represents. Since all variables in $P''$ are static, the node size can be determined from dataflow analysis; the node size may be an array of length $u$. As a data space is reached in $P''$, a new record is added to the list to hold the values of that data state.

A question arises when a loop is encountered as to how to store the data states from the many iterations; many natural data states may be produced at a single location. Should all iteration occurrences of natural data states be recoreded into the same subdata space from a particular execution? As a motivational answer, suppose there exists a program where 20% of the executions execute a particular location $10^6$ times, and on 80% of the executions the location is executed once. If all occurrences of natural data states at a location are entered into one value

[19] A variable is active for a path equivalence class if and only if it is active at some location on that path equivalence class. Then the active variables for a path equivalence class is just the union of each set of variables of each location in the path equivalence class.

[20] Assuming production of the path equivalence classes of $P''$.

distribution, the value distribution will be biased. Since the failure propagation algorithm perturbs on $f_l(\mathcal{B}_{l,1}(x))$, then the storage must contain enough information such that the data states encountered on the first execution of a location are differentiable from the succeeding data states of the location. The answer is to have a separate value distribution for the data states of the first iteration, and one value distribution for all succeeding data states of the location, similar to the Method I scheme of having a separate symbolic expression for each iteration of the loop. The failure propagation algorithm only perturbs the first natural data state at a location, so sampling only occurs from the value distribution representing the *first occurrence* natural data states. Infection analysis samples uniformly across all data states regardless of the iteration; however, sampling within a value distribution may not be uniform; it is according to the distribution.

At the end of an execution of $P''$, Method II produces a data object similar to Figure 4.3 (a). Pattern matching can be done on the output string produced by the "write" statements. In Figure 4.3 (a), each node can be thought of as representing a sequence of data states similar to $[\mathcal{A}_{1,1}(b), \mathcal{A}_{succ(1,1,b),1}(b), ..., \mathcal{A}_{succ(1,n,b),n}(b), ..., \mathcal{A}_{exit\_location,1}(b)]$ for some input $b$. Once comparison is made as to which path equivalence class was executed, the values included in the $u$-tuple belonging to dead variables are discarded(shown in Figure 4.3 (b) by the darkened spaces). Each record which was a $u$-tuple is now converted to a $z$-tuple. Then the values in the list can be added to the value distributions in the corresponding subdata spaces. The linked list is then deleted.

The method of storing data states in the value distribution is arbitrary; decision of the type of structure to use must be made. Two schemes are presented: encoding a data state to a single value for placement into a histogram[21], and storage of a node in the execution derived linked list in a value distribution linked list.[22] It is necessary that the $z$-tuple of values stored in a record in the linked list remains stored together in the value distributions.

The execution derived linked list may be stored in a different linked list, i.e., take the nodes from the linked list and place them in another linked list. So the value distributions at a subdata space will be contained in a linked list; in fact, two linked lists for each subdata space. One for the first occurrence data

[21] One expensive and infeasible method that accomplishes this is Gödel encoding [18]. It is solely mentioned for explanatory purposes. The values in the $z$-tuple are encoded into a single value that is placed in a histogram. A histogram is the structure used for storing the value distributions. The histogram contains the frequency counts of particular natural data states. For encoding, there must be consistent ordering of the variables within a subdata space. The infeasibility of Gödel encoding stems from it producing a value that exceeds most integer limits.

[22] Neither scheme is advised. This is an area for future research.

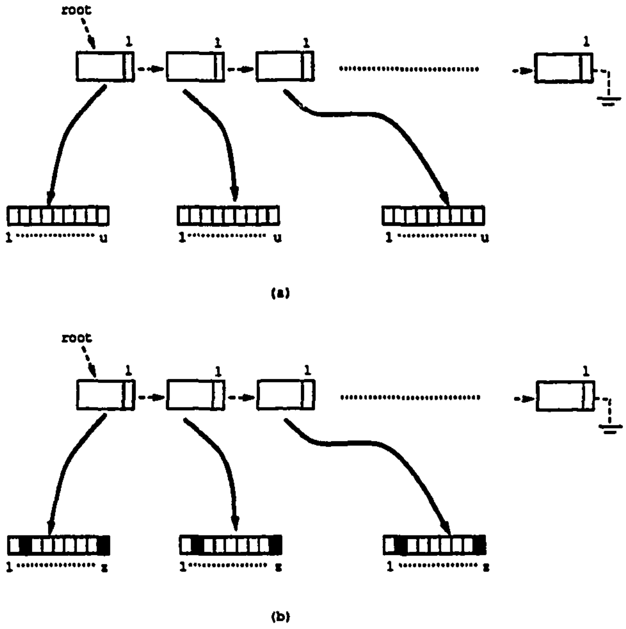


Fig. 4.3: Data structure representation for natural data states for Method II

states, and one for the remaining occurrence data states. Each node in the value distribution linked list should hold both the values and a frequency count of the number of times that particular data state combination occurred. Although it seems improbable that any particular data state occurs noticeably more frequently, especially as the number of active variables increases, [14] gives an interesting example showing that from $2^{15}$ possible data states, only 20 occurred out of a sample space size of $10^6$.

To implement this scheme, create a pointer for each type of occurrence at a subdata space, and initialize these to nil. Each time a data state not previously calculated during execution is discovered at a data space, create a new node in the list corresponding to its values with a frequency count of one. If the data state being added already exists, find the node for that data state in the list and increment its frequency count. Nodes with higher frequency counts can be moved to the head so that when sampling occurs later, values with highest probability can be found easily. This scheme saves in the overhead of an encoding/decoding scheme, however as the size of the list grows, the cost of searching during insertion grows. Also, the spatial requirements of storing all of the lists is prohibitive. For spatial reasons, this scheme is considered infeasible.

Again a mean-value analysis is given of the spatial requirements for Method II assuming encoding. Again, let $t = \mid T \mid + \mid TS \mid$, $\mu_t$ = the mean number of locations per path equivalence class, $\mu_r$ = the mean number of locations in a loop on a path equivalence class, and $\mu_a$ = the mean number of active variables per location. Then the mean number of value distributions is the same as the number of expressions for Method I: $\Theta(t(\mu_t - \mu_r)\mu_a + 2t\mu_r\mu_a)$.[23] If values are assumed to be stored in a histogram, the mean number of bins per distribution is $\mu_b$, where each bin requires 32-bits for the frequency counts, the space order required is $\Theta(4\mu_b t(\mu_t - \mu_r)\mu_a + 8\mu_b t\mu_r\mu_a)$ bytes. To get a feeling for this, assign values as follows: $t = 500$, $\mu_t = 100$, $\mu_j = 10$, $\mu_r = 10$, $\mu_a = 50$, and $\mu_b = 100$, This example requires 0.218 giga-bytes, a savings over Method I. The temporal requirements to get a natural data state from Method II ignoring the time involved in executing $P''$ many times and storing the natural data states is a function of the search and retrieve time in a value distribution. If constant search and retrieve time, $\sigma$, is assumed, then the mean time to get one complete natural data state is $\Theta(\sigma\mu_a)$. The mean-value analysis of Method II using the linked list still has $\Theta(t(\mu_t - \mu_r)\mu_a + 2t\mu_r\mu_a)$ value distributions, which means $\Theta(t(\mu_t - \mu_r)\mu_a + 2t\mu_r\mu_a)$ linked lists. The overhead in maintaining a list is high and not recommended.

Recapping, the algorithm needed for producing value distributions of natural data states at the subdata space abstraction level of Method II are:

1. Declare an empty linked list to exist with the size of each node large enough to hold all statically declared variables
2. Select an input point,
3. As each location is reached, create a new node in the list and store all values known in it,
4. At the end of an execution pattern match on the output string and determine the path equivalence class taken,
5. Delete all values in the linked list that do not correspond to the active variables of that path equivalence class,
6. Place the remaining values from each node into the value distribution scheme chosen,

[23] The 2 represents the two different type of iterations: first occurrences, and the remaining occurrences.

7. Delete the linked list and repeat the process.

In summation, Method II is also a high overhead scheme. However when a realistic algorithm of storing $n$-tuples of values is found, Method II will be practical. This is an area for further study.

#### 4.4.2.3 The Algorithm for Method III

Method III is a scheme which is temporally expensive yet has minimal spatial requirements. This algorithm is in general preferred, and if the program contains dynamic variables, it is the only one of the three algorithms that can be easily implemented to handle them. The algorithm is simple enough to be included into the steps of the main processes. Sections 4.5 and 4.6 show how to incorporate Method III into Process Propagation Analyzer and Process Infection Analyzer. The algorithm not only produces natural data states; it also automatically samples the data state it most recently created. So in effect Method III handles production and sampling of natural data states simultaneously. The algorithm follows:

1. set countl to zero,
2. take $P''$ and insert the necessary code to perform the algorithms of Process Propagation Analyzer or Process Infection Analyzer at the location,
3. select an input point and execute the augmented $P''$,
4. if this input point causes the location under analysis to be reached, increment countl,
5. execute $P''$ until countl $= n$,
6. when countl $= n$, divide count by countl yielding an infection estimate for this alternative,[24]
7. the previous six steps have produced an infection or propagation estimate; now repeat this process according to the algorithms of Chapter 3 for each semantic alternative or active variable.

[24] The infection estimate is a point estimator which is the sample mean of the number of infections, however to determine the confidence intervals in the estimate, the confidence interval of the frequency distribution for 95% confidence can be found with $p \pm w$, where $w = 2 \cdot \sqrt{(p \cdot (1-p)/n)}$ and $p =$ count $/n$ [11].

Unfortunately, this scheme requires the repeated execution of locations to produce natural data states. It may also cause performing executions that may not reach the desired location if $\phi$ is not previously partitioned. Choi *et al.* [12] present a scheme for using a hypercube for scheduling mutants during mutation analysis. A similar scheme may also be applied to the Method III implementation of Process Natural Data State Producer in order to save re-executing locations.

The temporal mean-value analysis of the seven step sequential algorithm above is as follows: let $\mu_e$ = the mean amount of time to execute a location, $\mu_l$ = the mean number of executions of the program before an execution where the location being analyzed is reached, $\mu_k$ = the mean number of locations per execution, and $\mu_p$ = the mean number of locations executed before this location. Then the mean amount of time to get one natural data state using Method II is $\Theta(\mu_e\mu_l\mu_k + \mu_p\mu_e)$ $= \Theta(\mu_e(\mu_l\mu_k + \mu_p))$. The spatial requirement for Method III is negligible.

### 4.4.3 Sampling from Value Distributions for Methods I and II

The method of sampling data states depends upon whether the value distributions or symbolic expressions are for data spaces or subdata spaces, and whether the process for failure propagation or infection analysis is requesting the states. Section 4.4.3 only applies to the value distributions and symbolic expressions created by Methods I and II.

#### 4.4.3.1 Method I Sampling

In the implementation of Method I, the symbolic expressions are at the subdata space abstraction level. The following algorithm for getting from symbolic expressions to natural data states is presented through example.

Suppose that the frequency with which path equivalence class $p$ is executed, $E_p$, is known for each path equivalence class going through some location $x$, and suppose sampling of data states is at location $x$ for either failure propagation analysis or infection analysis. And suppose that $n$ sampled data states are needed for analysis at data space $x$ and that data space $x$ has $k$ subdata spaces (so there are $k$ path equivalence elements that have location $x$ in their regular expressions). For failure propagation analysis at location $x$, generate

$$n \cdot E_p \quad \text{where } 1 \leq p \leq k$$

natural data states from the first occurrence symbolic expressions and first occurrence input conditions for this path equivalence class in each subdata space $p$ of data space $x$. Denote $\mu_{x,p}$ to be the mean number of expressions per active variable through location $x$ regardless of iteration occurrence for an element of path equivalence class $p$. For infection analysis, generate

$$n \cdot \frac{E_p}{1} \cdot \frac{1}{\mu_{x,p}}$$

natural data states uniformly from the first occurrence expressions and the input conditions for this path equivalence class in subdata space $p$ at location $x$. A proportion of data states representing the first iteration is chosen separately from the other iteration occurrence data states to reflect the proportion of each that will be seen in the dynamic environment of the program. This is automatically handled by Method III. After these are generated, then generate

$$n \cdot \frac{E_p}{1} \cdot (1 - \frac{1}{\mu_{x,p}})$$

natural data states by uniformly selecting expressions from the non-first occurrence expressions with the input conditions for this path equivalence class in subdata space $p$ at location $x$.[25]

#### 4.4.3.2 Method II Sampling

If the value distributions are for data spaces, then sample according to the distribution of the value distribution and according to the process's algorithm.[26] If the value distributions are for subdata spaces, the ensuing discussion explains how to sample across subdata space value distributions.

As an example, suppose that the frequency with which path equivalence class $p$ is executed, $E_p$, is known for each path equivalence class going through some location $x$, and suppose sampling of data states is at location $x$ for either failure propagation analysis or infection analysis. And suppose that $n$ sampled data states are needed for analysis at data space $x$ and that data space $x$ has $k$ subdata spaces. For failure propagation analysis at location $x$, sample

$$n \cdot E_p \quad \text{where } 1 \leq p \leq k$$

[25] If this location is not in a loop, there are only first occurrence expressions.

[26] Recall that infection analysis uses different data states for repeated locations than does failure propagation analysis.

natural data states from the first occurrence value distribution in each subdata space $p$ of data space $x$. Denote $\mu_{x,p}$ to be the mean number of times through location $x$ for an element of path equivalence class $p$. For infection analysis, sample

$$n \cdot \frac{E_p}{1} \cdot \frac{1}{\mu_{x,p}}$$

natural data states uniformly from the first occurrence value distribution in subdata space $p$ at location $x$. Then sample

$$n \cdot \frac{E_p}{1} \cdot (1 - \frac{1}{\mu_{x,p}})$$

natural data states uniformly from the non-first occurrence value distribution in subdata space $p$ at location $x$.

A problem occurs during sampling in Method II if a location was never reached during value distribution creation; the value distributions for the corresponding data space are empty. If a large number of program executions occurred during value distribution creation, either

1. the location is infeasible,
2. the location is feasible but rarely executed.

Unfortunately, case (1) can not be distinguished from case (2). So the best that can occur will be to either

1. re-execute $P''$ until this location starts executing; modify the value distribution creation routine to only store value distributions for this location, or
2. create artificial data states.

If (1) is attempted, and after a set time limit the location is still not reached, then artificial data states will need to be created.

## 4.5 Process Propagation Analyzer

Process Propagation Analyzer is a main process that performs the failure propagation algorithm. Input to this process is $P''$ and the natural data states from

Process Natural Data State Producer. Output from Process Propagation Analyzer is a program which when executed produces a failure propagation estimate for each active variable at each location in $P''$. The following pseudo-code shows a shell of the program outputted by Process Propagation Analyzer. This pseudo-code shell uses Method III value distribution creation and sampling.

```
procedure execute_unperturbed_code(input)
  location_1
  location_2
    :
    :
  location_n

procedure execute_perturbed_code_loc_1_active_variable_1(input)
    :
    :
procedure execute_perturbed_code_loc_1_active_variable_x(input)
    {assuming x active variables @location 1}
procedure execute_perturbed_code_loc_2_active_variable_1(input)
    :
    :
procedure execute_perturbed_code_loc_2_active_variable_z(input)
    {assuming z active variables @location 2}
procedure execute_perturbed_code_loc_n_active_variable_1(input)
procedure execute_perturbed_code_loc_n_active_variable_2(input)
procedure execute_perturbed_code_loc_n_active_variable_k(input)
    {assuming k active variables @location n}
  begin
    location_1
    location_2
      :
      :
    location_n
    arrived@location[n,k] := true
    if not(already_perturbed[n,k]) then
      begin
        perturb(data state@n active variable k)
        already_perturbed[n,k] := true
      end;
  :
```

```
 :
 end;


begin main
  {set count and problem arrays to zero }
  for x := 1 to N { N is number of program locations }
    get input
    call execute_unperturbed_code(input)
    for i := 1 to m { m is number of perturbations to occur
                      for one active variable
                      for one data state }
      for each active variable k ∈ $\mathbf{A}_i$
        arrived@location[i,k] := false
        already_perturbed[i,k] := false
        call execute_perturbed_code_loc_i_active_variable_k(input)
        if arrived@location[i,k] then
          count[i,k] ++
        if arrived@location[i,k] and (output(execute_unperturbed_code) <>
        (output(execute_perturbed_code_loc_i_active_variable_k)) then
          problem[i,k] ++


  for i := 1 to m
    for each active variable k ∈ $\mathbf{A}_i$
      writeln('prop. est. for location i active var. k =', problem[i,k]
                       / count[i,k])
        {assuming count[i,k] <> 0}
end main
```

It is now shown that the pseudo-code shell is equivalent to:

$$\mathrm{F}_{l,a} = \Pr[\text{pseudo-failure on input } x \mid \textit{perturbed}(l,a,x)]$$

where

$$\textit{perturbed}(l,a,x) = \begin{cases} \mathbf{T} & \text{iff } a \in f_l(\mathfrak{B}_{l,1}(x)) \text{ and } \textbf{function} \text{ perturb(a)}^3 \text{ has executed} \\ & \text{on input } x \\ \mathbf{F} & \text{otherwise} \end{cases}$$

Satisfaction of "*perturbed*$(l, a, x) = \mathrm{T}$ iff $a \in f_l(\mathcal{B}_{l,1}(x))$ and **function** perturb(a) has executed on input $x$" occurs with the following code segment:

```
location_l
arrived@location[l,a] := true
if not(already_perturbed[l,a]) then
  begin
    perturb(data state@l active variable a)
    already_perturbed[l,a] := true
  end;
```

where (already_perturbed[l,a]) is set to false before call execute_perturbed_code_loc_i_active_variable_k(input) is called to assure perturbation only on the first data state. The total number of pseudo-failures is found by:

```
if arrived@location[l,a] and (output(execute_unperturbed_code) <>
(output(execute_perturbed_code_loc_l_active_variable_a)) then
  problem[l,a] ++
```

The total number of perturbations occurring on the first occurrence data state for location $l$ and active variable $a$ is found by:

```
if arrived@location[l,a] then
  count[l,a] ++
```

So the point estimator which is the failure propagation estimate is:

$$\Pr[\text{pseudo-failure on input } x \mid \textit{perturbed}(l,a,x)] = \frac{\text{problem[l,a]}}{\text{count[l,a]}} \quad \blacksquare$$

The $O$-notation for this implementation of the algorithm for a program of $m$ locations in straight-line is as follows [15]: Suppose that $n$ data states are to be used at each location, and that there are $\mu$ active variables on average at each location. Then this implementation has order $O(n\mu m^2 + n\mu)$, hence it is quadratic in the number of locations executed, excluding the locations to perform the perturbations.[27]

[27] $n\mu$ is the number of locations executed from the non-perturbed copy.

## 4.6 Process Infection Analyzer

Process Infection Analyzer is a main process which performs the algorithm for finding infection estimates. Input to this process is $P''$ and the natural data states from Process Natural Data State Producer. Output from this process is a program which when executed produces an infection estimate for each location in $P''$ and each semantic alternative. The following pseudo-code shows a shell of the program that this process outputs. This shell uses Method III value distribution creation and sampling.

It should be noted that the technology involved in Process Infection Analyzer is in most part the same as mutation testing[28][16, 17]. The difference is in the data collected. In mutation testing, the input points are evaluated. In infection analysis, the location with the semantic alternative is evaluated.

```
procedure execute_locations(input)
begin
  times@location[1] ++
  if ((data state after location_1) <>
     (data state after semantic_alternative_1(location_1)) then
       counter_1[1] ++
       :
       :
       :
       :
  if ((data state after location_1) <>
     (data state after semantic_alternative_k(location_1)) then
       counter_1[k] ++
  location_1
     :
     :
     :
  location_2
     :
     :
  times@location[n] ++
  if ((data state after location_n) <>
     (data state after semantic_alternative_1(location_n)) then
```

[28] Mutation testing is a way of determining test data adequacy by seeing if the test data catches faults injected into the code.

```
      counter_n[1] ++
      :
      :
      :
      :
  if ((data state after location_n) <>
    (data state after semantic_alternative_k(location_n)) then
      counter_n[k] ++
  location_n
end


begin main
  for k := 1 to number_of_semantic_alternatives@location_i
    counter_i[k] := 0
  for i := 1 to n {n is number of locations }
    times@location[i] := 0
  for z := 1 to N {N is number of inputs for confidence interval }
    get input
    call execute_locations(input)
  for i := 1 to n
    for k := 1 to number of semantic_alternatives@location_i
      writeln('inf. est. for ',k, counter_i[k]/times@location_i)
           {assuming times@location_i <> 0}
end main
```

It is now shown that the pseudo-code shell is equivalent to:

$$I_{l,a} = \Pr[\mathit{infected}'(l,a,x) \mid \mathfrak{B}_l(x) \neq \emptyset]$$

where

$$\mathit{infected}'(l,a,x) = \begin{cases} \mathsf{T} & \text{iff } \exists y \in \mathfrak{B}_l(x)\ f_l(y) \neq f_a(y) \\ \mathsf{F} & \text{otherwise} \end{cases}$$

To verify that $\mathfrak{B}_l(x) \neq \emptyset$,

```
times@location[l] ++
```

is used so then the point estimator is a function of the number of arrivals at location $l$. Satisfaction of $\mathit{infected}'(l,a,x) = \mathsf{T}$ iff $\exists y \in \mathfrak{B}_l(x)\ f_l(y) \neq f_a(y)$ occurs with

```
if ((data state after location_l) <>
  (data state after semantic_alternative_a(location_l)) then
    counter_l[a] ++.
```

So then the point estimator for the infection estimate for location $l$ and semantic alternative $a$ is:

$$\Pr[\mathit{infected}'(l,a,x) \mid \mathcal{B}_l(x) \neq \emptyset] = \frac{\text{counter\_l[a]}}{\text{times@loc\_l}}$$

■

The $\mathcal{O}$-notation for this implementation of the algorithm for a program of $m$ locations in straight-line is as follows: Suppose that $n$ data states are to be used at each location, and that there are $\mu$ semantic alternatives on average at each location. Then this implementation has order $\mathcal{O}(n\mu m)$, hence it is linear in the number of locations executed.

Using an idea from Choi *et al.* [12], the above sequential algorithm could parallelized as follows: assign one processor to execute the location as it currently stands. So it is essentially performing two roles: as the keeper of the correct result for each location, and as the natural data state producer. For each location under analysis, assign each semantic alternative to a different processor. Thus each processor with a semantic alternative gets two messages from the processor with the correct location: the first contains the input natural data state to use, and the second is the output data state from the original location. The semantic alternative processors can then determine if an infection occurs and keep a record of the number of infections. The program under analysis can be stepped through in such a manner to greatly decrease the time to perform infection analysis. Any speed up achieved, however, will decrease the order of the implementation to no less than $\mathcal{O}((n\mu m)/p)$, where $p$ is the number of processors. An analytical scheme for expression infection estimation is presented at the end of Section 4.6.

Process Infection Analyzer requires a sub-process to produce the semantic alternatives for each location; Section 4.6.1 explains what semantic alternatives should be produced. There are two types of infection analysis performed: one for boolean predicates and one for arithmetic expressions. Different rules are invoked for producing semantic alternatives depending on the location type.

| *semantic alt.* | *semantic alternative description* |
|---|---|
| 1. | $a - b$ |
| 2. | $a * b$ |
| 3. | $a/b$ |
| 4. | $b_1 + b$ over all active variables for $b_1$ at location of a+b |
| 5. | $a + a_1$ over all active variables for $a_1$ at location of a+b |
| 6. | $const + b$ |
| 7. | $a + const$ |
| 8. | $(const_k \cdot a^k + const_{k-1} \cdot a^{k-1} + .. + const_0) + b$ where $-\infty < const_k, const_{k-1}, ..., const_0 < \infty$ |
| 9. | $a + (const_k \cdot b^k + const_{k-1} \cdot b^{k-1} + .. + const_0)$ where $-\infty < const_k, const_{k-1}, ..., const_0 < \infty$ |

Table 4.3: Semantic Alternatives for the expression $(a + b)$

### 4.6.1 Expression Infection Analysis

Expression infection analysis finds an infection estimate for an assignment statement in $P''$ in simple expression form. For expressions, class $\mathcal{F}$ is limited to the following fault types, all of which are single changes to a location:

1. a wrong variable substitution,
2. a variable substituted for a constant,
3. a constant substituted for a variable,
4. expression omission,
5. a variable that should have been replaced by a polynomial of degree $k$,
6. and a wrong operator.

The breadth of the type of faults defined in $\mathcal{F}$ determines the power of Process Infection Analyzer.

These six fault types are the ones that this thesis considers; other applications or external knowledge of the developmental environment may require changes in $\mathcal{F}$. As an example of $\mathcal{F}$ applied to the arithmetic expression $a + b$, the set of semantic alternatives derived from $\mathcal{F}$ are in Table 4.3.[29] Similar tables can be derived for simple expressions with other operators.

[29] The range on the values to be substituted as constants for fault 6 and fault 7 is limited to the discrete values for variable "a" which have the highest probabilities of occurring in the value distribution preceding the location.

### 4.6.2 Predicate Infection Analysis

Predicate infection analysis is performed on each predicate. The fault classes in $\mathcal{F}$ for predicate infection analysis are:

1. substituting a wrong variable,
2. exchanging and and or, and
3. substituting a wrong equality/inequality operator.

For loops, exchanging equality/inequality operators can quickly cause infinite loops, so care must be taken to minimize the potential. Therefore a time limit is placed in step 3 of the infection algorithm. Even though a particular semantic alternative may be in the class $\mathcal{F}$, if it is predetermined that termination does not occur when the semantic alternative is substituted, then the semantic alternative should be skipped. The intent of infection analysis is to mimic "common" faults. Any fault causing an infinite loop should not be considered "common" by the closely correct criterion, unless it is at a rarely executed location. Notice again that the class of faults defined for predicate infection analysis represent single changes to a location, not multiple changes. This definition of $\mathcal{F}$, again, is subject to change under various conditions. Such conditions are expected due to strange software phenomenon however these conditions are currently not known.

As an example of predicate infection analysis, for the predicate ((a < b) and (c)), where a and b have numeric values and c has a boolean value, the semantic alternatives are given in Table 4.4. Most of the semantic alternatives result in high infection estimates. The semantic alternatives which may produce lower estimates are 1, 6, 7, and 8. The viruses that are produced from predicate faults are in the program counter. Such faults cause a wrong path to be taken. The notion of taking a wrong path yet producing the correct result is termed *coincidental correctness* [1]. A missing path can be partially detected by failure propagation analysis.[30]

#### 4.6.2.1 Analytical Expression Infection Analysis

This section explains an alternative scheme for determining the expression infection estimates for the nine semantic alternatives of Table 4.3. For the first semantic

[30] Partial detection can be expanded to near total detection if propagation analysis is performed on every declared variable at each location. The reason total detection is not possible is because the effect of the missing path my be to some variable which has not yet been declared.

| *fault* | *fault description* |
|---|---|
| 1. | $(a \leq b)$ and $c$ |
| 2. | $(a \geq b)$ and $c$ |
| 3. | $(a > b)$ and $c$ |
| 4. | $(a = b)$ and $c$ |
| 5. | $(a \neq b)$ and $c$ |
| 6. | $((b_1 < b)$ and $c)$ over all declared active boolean variables $b_1$ |
| 7. | $((a < b_1)$ and $c)$ over all declared active boolean variables $b_1$ |
| 8. | $(a < b)$ or $c$ |

Table 4.4: Semantic Alternatives for the predicate ((a < b) and c)

alternative of Table 4.3, the infection estimate is $1 - \Pr[a + b = a - b]$. The probability $\Pr[a + b = a - b]$ can be solved for all (a,b) pairs using a scheme such as the Mathematica [19] Solve directive. Solve produces the (a,b) pairs where $a + b = a - b$. After these pairs are found, determine the frequency of each pairs occurrence in the data space value distributions preceding the location; this then is $\Pr[a + b = a - b]$. Only when $b = 0$ does $\Pr[a + b = a - b]$, hence $\Pr[a + b = a - b] = \Pr[b = 0]$. If in the preceding data space to the expression there exists a high preponderance of zeros in $b$'s value distribution, then $\Pr[a + b = a - b]$ is high causing a low infection estimate. Similar tactics also yield infection estimates for semantic alternatives 2 and 3 of Table 4.3.

For semantic alternatives 4 and 5, the infection estimate is high unless $b_1$ and $a$ have identical values in the preceding data state, i.e., data states (a,b) such as (1,1), (2,2), (3,3), etc. Infection estimates for semantic alternatives 4 and 5 are found by $1 - \Pr[b_1 = a$ in the preceding data state$]$. To estimate $\Pr[b_1 = a$ in the preceding data state$]$, sample the natural data states and find the frequency of $(x, x)$ value pairs in the value distribution. One minus this frequency is the infection estimate.

The infection estimate for semantic alternative 6 is $1 - \Pr[a = \mathit{constant}]$. $\Pr[a = \mathit{constant}]$ is found by finding the point or points (if there are several) with the highest relative frequency in the value distribution for $a$. Subtract this highest relative frequency for this bin from 1.0 yielding the minimum probability that this semantic alternative causes infection; this is the minimum infection estimate. Similar tactics also yield an infection estimate for semantic alternative 7.

Finding infection estimates for semantic alternatives 8 and 9 is more difficult. When a polynomial of degree $k$ replaces a variable, an explosion in the actual number of semantic alternatives each semantic alternative expression represents occurs due to values the constants can be assigned$(-\infty < \mathit{const}_0, \mathit{const}_1, ..., \mathit{const}_k < \infty)$.

If the degree of the substituted polynomial is limited to 1, then only semantic alternatives of the form $(const_1 \cdot a + const_0)$ need be considered for semantic alternative 8, and $(const_1 \cdot b + const_0)$ for semantic alternative 9. The expression $(const_1 \cdot b + const_0 = b)$ is true iff

$$b = \frac{const_0}{(1 - const_1)}$$

The truth of this equation is a function of $b$, $const_1$, and $const_0$. In general, replacing a variable by a polynomial should produce an infection estimate of approximately 1.0, however the analysis should still occur if polynomial substitutions are considered as potential faults.[31] If consideration is limited at the three dimensional value space for the two constants $const_0, const_1$ over $-\infty$ to $\infty$, and at the highest frequency values of $b$ which are shown in Figure 4.4, it becomes likely that the number of times that the predicate is true is execedingly small; hence infection always occurs.[32] Infection for semantic alternatives 8 and 9, where the degree of the replacing polynomial is 1, is then

$$1 - \mathrm{Pr}[\frac{\text{number of times predicate is true}}{\text{size of 3-D space}}]$$

which is shown in Figure 4.4;

$$\mathrm{Pr}[\frac{\text{number of times predicate is true}}{\text{size of 3-D space}}]$$

is the proportion of dark points in the darkened box relative to the total space of points in the box. If we consider the case of a polynomial of degree $x$ with $x$ solutions, then the probability of no infection occurring is just

$$\mathrm{Pr}[\frac{x}{(2 \cdot \mathrm{maxint})^x}]$$

which is infinitesimally small. Note that the total number of values for $b$ is not infinite in Figure 4.4; consideration only occurs for values of $b$ that are more likely, hence the tick marks in Figure 4.4 represent the more likely values of $b$. These values can be found in the value distribution at $ds\mathcal{P}_{\mathrm{a+b}}$.

[31] Particularly low order polynomials.

[32] Actually, [-maxint..maxint] is the range for $const_1$ and $const_0$.

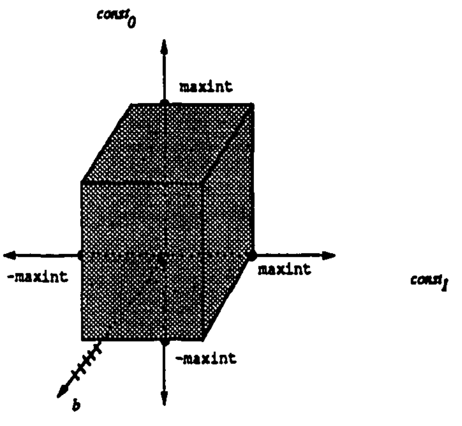


Fig. 4.4: 3-D space for a one-degree polynomial replacement for variable

| *location* | *infection estimate* | *propagation estimate* |
|---|---|---|
| 1. | $\min_i\{I_{1,a_i}\}$ | $\min_k\{F_{1,a_k}\}$ |
| 2. | $\min_i\{I_{2,a_i}\}$ | $\min_k\{F_{2,a_k}\}$ |
| 3. | $\min_i\{I_{3,a_i}\}$ | $\min_k\{F_{3,a_k}\}$ |
| 4. | $\min_i\{I_{4,a_i}\}$ | $\min_k\{F_{4,a_k}\}$ |
| : | : | : |
| ↓ | ↓ | ↓ |

Table 4.5: Template of Minimum Data from PIA

## 4.7 Infection Analysis Output and Failure Propagation Analysis Output

Propagation and infection analysis produces a large number of infection and failure propagation estimates for each location. The output from Process Propagation Analyzer and Process Infection Analyzer is similar to that shown in Table 4.5.

For certain applications, more information is needed than that shown in Table 4.5. Almost all of the estimates are lost when only the minimum estimate is provided. Saving each estimate for each semantic alternative and each active variable requires a large amount of space. The template in Table 4.6 shows complete propagation and infection analysis results.

| *location* | *infection estimates* | | | | | | *propagation estimates* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $\rightarrow$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $\rightarrow$ |
| 1. | $I_{1,a_1}$ | $I_{1,a_2}$ | $I_{1,a_3}$ | $I_{1,a_4}$ | $I_{1,a_5}$ | $\rightarrow$ | $\mathrm{F}_{1,a_1}$ | $\mathrm{F}_{1,a_2}$ | $\mathrm{F}_{1,a_3}$ | $\mathrm{F}_{1,a_4}$ | $\mathrm{F}_{1,a_5}$ | $\rightarrow$ |
| 2. | $I_{2,a_1}$ | $I_{2,a_2}$ | $I_{2,a_3}$ | $I_{2,a_4}$ | $I_{2,a_5}$ | $\rightarrow$ | $\mathrm{F}_{2,a_1}$ | $\mathrm{F}_{2,a_2}$ | $\mathrm{F}_{2,a_3}$ | $\mathrm{F}_{2,a_4}$ | $\mathrm{F}_{2,a_5}$ | $\rightarrow$ |
| 3. | $I_{3,a_1}$ | $I_{3,a_2}$ | $I_{3,a_3}$ | $I_{3,a_4}$ | $I_{3,a_5}$ | $\rightarrow$ | $\mathrm{F}_{3,a_1}$ | $\mathrm{F}_{3,a_2}$ | $\mathrm{F}_{3,a_3}$ | $\mathrm{F}_{3,a_4}$ | $\mathrm{F}_{3,a_5}$ | $\rightarrow$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\rightarrow$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\rightarrow$ |
| $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\rightarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\rightarrow$ |

Table 4.6: Template of Accumulated Data from PIA

## 4.8 Relating Propagation and Infection Estimates to the Thesis Objective

Propagation and infection estimates relate directly to the thesis goal: "*determining where a fault can easily hide.*" A location which has small propagation and infection estimates is a location which may easily hide a fault. It also is the case that a location which is rarely executed, may easily hide a fault. So formally, for a location $l$,[33]

$$\exists k\ \mathrm{F}_{l,a_k} \approx 0.0 \vee \exists k\ I_{l,a_k} \approx 0.0 \vee E_l \approx 0.0 \Longrightarrow$$
$$\text{location } l \text{ can easily hide a fault} \tag{4.1}$$

Similarly,

$$\neg\exists k\ \mathrm{F}_{l,a_k} \approx 0.0 \wedge \neg\exists k\ I_{l,a_k} \approx 0.0 \wedge \neg(E_l \approx 0.0) \Longrightarrow$$
$$\text{location } l \text{ can not easily hide a fault} \tag{4.2}$$

Many locations may satisfy equation 4.1. To determine those locations which can more easily hide a fault than others relative to $\Delta$ even if they both satisfy equation 4.1, define a parameter $\zeta_l$ as the ability of a location to hide a fault. The larger $\zeta_l$, the greater this ability. Thus:

$$\zeta_l = w_1 \cdot \sum_k f(k, \mathrm{F}_{l,a_1,\ldots}) + w_2 \cdot \sum_k i(k, I_{l,a_1,\ldots}) + w_3 \cdot e(l) \tag{4.3}$$

where

$$f(k, \mathrm{F}_{l,a_1,\ldots}) = \begin{cases} 1 & \mathrm{F}_{l,a_k} \approx 0.0 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

[33] $\approx 0.0$ denotes on the order of $10^{-4}$ which was determined arbitrarily; $E_l$ denotes the execution rate of location $l$.

and

$$i(k, I_{l,a_1,\ldots}) = \begin{cases} 1 & I_{l,a_k} \approx 0.0 \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

and

$$e(l) = \begin{cases} 1 & E_l \approx 0.0 \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

and $w_1, w_2, w_3$ are weights associated with the importance of the three estimates. It is currently unclear which of these three sets of information is more important, hence $w_1, w_2$, and $w_3$ are left as parameters. It is also not clear whether a linear combination of $f$, $i$, and $e$ is appropriate, versus an exponential combination or some other function. Clearly, the size of $|\mathsf{A}_l\ |$ and $\mathcal{F}_l$ must play a role in $w_1$ and $w_2$.

## 4.9 Conclusions

This chapter details the main and auxiliary processes of PIA as well as alternative schemes for performing them. As more precise or computationally cheaper algorithms are found, the requirements on the state of the input program given in the introductory paragraphs of this chapter may be relaxed. It appears, although, that the closely correct criterion will always in some form exist however its definition may be relaxed. If in the future it becomes temporally trivial to perform propagation and infection analysis, then the closely correct criterion may be discarded. This currently appears unlikely.

A brief summary of the requirements of the processes presented in this chapter follows:

**Process Simplify** is invoked for creating more refined estimates of failure propagation at the subexpression level. It exists for better quality in the failure propagation estimate. If implemented, it can be performed manually and this process is neither spatially nor temporally expensive.

**Process Abstraction Analyzer** isolates the locations that the methodology will be applied to. This process is trivial; it can be performed manually and is trivial.

**Process Dataflow Analyzer** produces the set of active variables at each location that the failure propagation algorithm will be applied to. This process can be performed manually for small programs to determine which variables are active; this process is trivial for straight-line programs; dataflow algorithms can be applied for automating the process. This process is neither spatially nor temporally expensive.

**Process Natural Data State Producer** produces the natural data states that the failure propagation implementation and infection implementation need. This process can be incorporated into Process Propagation Analyzer and Process Infection Analyzer as shown in the pseudo-code shells; this makes this process temporally expensive; otherwise if a storage method is chosen it is spatially prohibitive.[34]

**Process Propagation Analyzer** produces the failure propagation estimates. When performed computationally, it is temporally expensive, however not overwhelmingly so; Process Propagation Analyzer builds the pseudo-code shell. It is quadratic in the number of locations executed assuming Method III value distribution creation. The main temporal requirements are in executing the shell, not creating it. This process has minimal spatial needs. Automating the creation of the pseudo-code shell is recommended, however this can be performed manually.

**Process Infection Analyzer** produces the infection estimates. This process will almost certainly be the hardest process to implement, since it requires many semantic alternatives for each location. This process is not spatially intensive, however it is temporally expensive in terms of creating the pseudo-code shell. It is linear in the number of locations executed assuming Method III value distribution creation. Creation of this shell will have the highest temporal costs, however dynamic temporal costs will not be as overwhelming. Automating the creation of this pseudo-code shell is recommended. No tool currently exists to perform the automation.

[34] Using a multi-processor machine with Method III natural data state production appears to be a practical approach to the space and timing problems of propagation and infection analysis. Any gains in performance, however, will be directly tied to the number of processors, hence the decrease in costs is linearly proportional to the number of processors.

# References

[1] LEE J. WHITE AND EDWARD I. COHEN. A domain strategy fo computer program testing. *IEEE Transactions on Software Engineering*, SE-6:pp. 247–257, May 1980.

[2] BOEING COMPUTER SERVICES COMPANY SPACE AND MILITARY APPLICATIONS DIVISION. *Integrated Verification and Testing System IVTS System Maintenance Manual Volume V - Data-Flow Analysis.* August 1983.

[3] LEON J. OSTERWEIL AND LLOYD D. FOSDICK. Dave-a validation error detection and documentation system for fortran programs. *Software Practice and Experience*, pp. 473–486, October-December 1976.

[4] WILLIAM E. HOWDEN. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, July 1977.

[5] MICHAEL KARR AND DAVID B. LOVEMAN III. Incorporation of units into programming languages. *Communication of the ACM*, vol. 21:pp.385–391, May 1978.

[6] BODGAN KOREL. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9), September 1988.

[7] BODGAN KOREL. The program dependence graph in static program testing. *Information Processing Letters*, January 1987.

[8] LANUSZ W. LASKI AND BODGAN KOREL. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[9] F. H. MARTIN. Hal/s - the avionics programming system for shuttle. *Proceedings AIAA Conference on Computers in Aerospace*, pp. 308–318, November 1977.

[10] HARLAN D. MILLS. *Software Productivity.* Little, Brown, and Company, 1983.

[11] STEVE PARK. *Lecture notes on simulation, Unpublished.* College of William and Mary, August 1988.

[12] BYOUNGJU CHOI AND ADITYA P. MATHUR AND BRIAN PATTISON. P$^M$othra: scheduling mutants for execution on a hypercube. *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, pp. 58–65, December 1989.

[13] TERRENCE W. PRATT. *Programming Languages Design and Implementation Second Edition.* Prentice-Hall Inc., 1984.

[14] PETER W. PROTZEL. Automatically generated acceptance test: a software reliability experiment. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 196–203, July 1988.

[15] ELLIS HOROWITZ AND SARTAJ SAHNI. *Fundamentals of Computer Algorithms.* Computer Science Press, Inc., 1978.

[16] RICHARD A. DEMILLO AND RICHARD J. LIPTON AND FREDERICK G. SAYWARD. Hints on test data selection: help for the practicing programmer. *Computer*, pp. 34–41, April 1978.

[17] TIMOTHY A. BUDD AND RICHARD A. DEMILLO AND RICHARD J. LIPTON AND FREDERICK G. SAYWARD. Theoretical and empirical studies on using program mutations to test the functional correctness of programs. *Conference Record 7th Annual ACM Symposium on Principles of Programming Languages*, pp.220–233, January 1980.

[18] MARTIN D. DAVIS AND ELAINE J. WEYUKER. *Computability Complexity and Languages.* Academic Press, 1983.

[19] STEPHEN WOLFRAM. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, 1988.

# Chapter 5

# Propagation and Infection Applications

Chapter 5 introduces application area models for propagation and infection analysis. The quantification of where a fault can easily hide may be used in probable correctness, software reliability, software testing, program debugging, and testing complexity. Research disciplines concerned with software quality can most likely either directly or indirectly benefit from propagation and infection analysis.

Section 5.1 introduces additional definitions and algorithms for the application area models of Chapter 5. Section 5.2 describes additional processes and mentions processes already introduced in Chapter 4 which need enhancements for these application models. The remaining sections of Chapter 5 are strictly devoted to the models.

## 5.1 Terminology and Definitions

Section 5.1 introduces three new entities: the latent failure rate, execution estimate, and dispersion histogram. The latent failure rate estimates the minimum failure probability of some entity at a particular abstraction level, the execution estimate is the probability of reaching an entity at some abstraction level(typically at path levels), and the dispersion histogram reveals which input points are believed to be more likely to reveal faults.

### 5.1.1 Latent Failure Rate

Section 5.1.1 introduces a method of estimating what the minimum failure rate will be for a program fragment depending upon the abstraction level. This section introduces a structure-based method of estimating the minimum failure rate. A black-box model for estimating the minimum failure rate of a program is to use the observed number of failures per number of executions. Another black-box scheme for estimating the minimum failure probability is Bayesian estimation [24]. The maximum failure rate of a fault in a program is 1.0. For several applications it is more useful to determine what the minimum impact on the failure rate will be for "any" fault. It, however, is not possible to make such a grandiose statement; limitations must to be placed on the interpretation of "any".

In order to measure this minimum failure probability, the conditions necessary for a program to fail are again reviewed. A fault must be reached, an infection must occur, and the infection must propagate. When these three occur, failure results, and the product of the frequency with which these three occur is the failure rate. The definitions from Chapter 2 for execution rate, propagation rate, and infection rate, combined with the assumption that location $x$ has the only fault in program $P$ can be related to the observed failure rate by:

$$\begin{aligned} \lambda_P &= \mathit{execution\ rate} \cdot \mathit{infection\ rate} \cdot \mathit{propagation\ rate} \qquad (5.1) \\ &= \Pr[\mathcal{B}_x(y) \neq \emptyset] \cdot \\ &\quad \Pr[\mathit{infected}(x,y) \mid \mathcal{B}_x(y) \neq \emptyset] \cdot \\ &\quad \Pr[\text{program fails on input } y \mid \mathit{infected}(x,y)] \qquad (5.2) \end{aligned}$$

Equation 5.2 is specific to one particular fault. If there are other faults in program $P$, this equation may be incorrect. So an attempt to produce a formula for the minimum failure probability should probably contain as parameters the infection estimates, propagation estimates, and execution rates.

The latent failure rate is the term used for the quantification of this minimum impact; the *latent failure rate* of a location $l$ is defined as the probability that if a fault exists at location $l$, and location $l$ is reached, failure occurs. In the formulae presented in Section 5.1.1 for measuring the latent failure rate, the execution rate is omitted. This separates the potential of a failure occurring from the potential of a particular abstraction level member from being executed. This ensures that low execution rates will not overshadow the cancellation occurring when infection does not occur and propagation does not occur.

As previously mentioned, restrictions are placed upon the interpretation of "any". The restriction placed on "any" will be the alternative class $\Delta$. Define the *alternative class*, $\Delta$, to be the class of impacts on the computation of the program which is defined by

1. $\mathcal{F}$,
2. the perturbation function parameters, and
3. the perturbing distribution.

Hence formulae for measuring the minimum failure probability at a location $l$ which represents the minimum probability of both infection and propagation occurring at a location is a function of $\Delta$ and is conditioned on reaching $l$.

The dependence of the latent failure rate on the propagation estimate should be clear. However the inclusion of the infection estimate in the latent failure rate may not be clear. It may seem defendable to assume that the infection estimate will be large. There are, however, faults that affect very few input points. As an example, replace `b := b mod 1000000` with `b mod 1000001`, where the distribution of values of `b` is in the range [0..1000000]. For values of `b` less than 1000000 no infection occurs. Another example is the replacement of `a := (a * 100)` with `if a≠1 then a:=(a*100) else a:=(a*10)`. In this example, only one value for `a` causes an infection. Hence the infection estimate plays an important role in representing faults which rarely infect. Ignoring the infection estimate may produce a drastically over-estimated latent failure rate.

Section 5.1.1 provides formulae for estimating the latent failure rate. A latent failure rate can be found for each level of abstraction. The distinguishing characteristic among the various methods of measuring a latent failure rate is how conservative an estimate the scheme produces, where scheme "A" is said to be more *conservative* than scheme "B" if for all latent failure rates scheme "A" produces, the estimates are lower than for scheme "B" for the same input parameters.

#### 5.1.1.1 Latent Failure Rate Measurement Assuming Independence

The first attempt at quantifying the minimum failure probability is a direct result of equation 5.2. Equations 5.3 - 5.5 assume independence between input points which propagate and infect, i.e., it is independent whether points which propagate also infect. So the latent failure rate for a location $l$ assuming independence is:

$$lfr_l = \min_k[I_{l,a_k}] \cdot \min_z[\mathrm{F}_{l,a_z}] \tag{5.3}$$

Equation 5.3 can be generalized to a path equivalence class $l$ with:

$$lfr_l = \min_{i,j}[\min_k[I_{j,a_k}] \cdot \min_z[\mathrm{F}_{i,a_z}]] \tag{5.4}$$

where locations $i$ and $j$ are on path equivalence class $l$. Notice that equation 5.4 has no requirement that location $j$ is a successor location of location $i$. Since it is typical to think of infection occurring before propagation, equation 5.4 is a bit peculiar. Hence equation 5.5 takes this into account producing a less conservative measurement:

$$lfr_l = \min_{i,j}[\min_k[I_{j,a_k}] \cdot \min_z[\mathrm{F}_{i,a_z}]] \quad \text{where } i \text{ precedes } j \tag{5.5}$$

#### 5.1.1.2 Latent Failure Rate Measurement Assuming Non-Independence

Section 5.1.1.2 describes a more conservative approach than presented by equations 5.3 - 5.5. It does not assume that those points which infect will propagate. It considers a class of input points referred to as non-propagators. A ***non-propagator*** is an input point which will cause an infection but will not propagate. Equations 5.6 - 5.8 subtract the proportion of non-propagators from the infection estimate, leaving an estimate of the proportion of the input points that will infect and propagate. This non-independence approach is more conservative than the independence approach; applications will later be mentioned which rely on having the most conservative estimate of the minimum failure probability. This is crucial for these applications.

The latent failure rate for a location $l$ assuming non-independence is measured by:

$$lfr_l = \min_k[I_{l,a_k}] - (1 - \min_z[\mathrm{F}_{l,a_z}]) \tag{5.6}$$

Equation 5.6 effectively removes the proportion of input points which for at least one perturbed active variable did not propagate. Equation 5.6 can be generalized to a path equivalence class. The "most conservative" measure of the latent failure rate for a path equivalence class $l$ is given by:

$$lfr_l = \min_{i,j}[\min_k[I_{j,a_k}] - (1 - \min_z[\mathrm{F}_{i,a_z}])] \tag{5.7}$$

where $i$ and $j$ are locations represented in path equivalence class $l$. If the same requirement from equation 5.5 is made for equation 5.7, the latent failure rate measure becomes:

$$lfr_l = \min_{i,j}[\min_k[I_{j,a_k}] - (1 - \min_z[\mathrm{F}_{i,a_z}])] \quad \text{where } i \text{ precedes } j \tag{5.8}$$

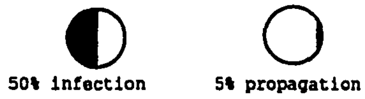


Fig. 5.1: High Infection, Low Propagation probabilities

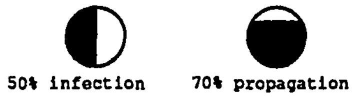


Fig. 5.2: High Propagation, High Infection probabilities

Equation 5.7 uses the lowest failure propagation estimate for any location $i$ in path equivalence class $x$ and the lowest infection estimate for any location $j$ in $l$. Equation 5.8 does also provided that location $i$ precedes location $j$.

Both equations 5.6, 5.7, and 5.8 inherit the potential for producing negative latent failure rates. This is due to the subtraction of the percentage of "nonpropagators". Although a latent failure rate estimate which is less than or equal to zero is useless as a probability, it contains useful information. A non-positive latent failure rate says that the abstraction level location $l$ may potentially (worst case) always produce the correct output even when there is a fault in a location on it.

Experimental results have shown that $\min_k[I_{j,a_k}]$ and $\min_z[F_{i,a_z}]$ values are in the majority of cases usually between 0.1 and 1.0.[1] Figure 5.1 and Figure 5.2 illustrate the difference which occurs between a positive and non-positive latent failure rate in equations 5.6 - 5.8. In Figure 5.1, the infection probability is 0.5, whereas the propagation probability is 0.05; this example produces a latent failure rate of -0.45. Notice in Figure 5.1 that the proportion of "infectors"(black); hence a negative latent failure rate. can easily be placed into the proportion of nonpropagators(white). So in the worst case, those points which propagate are those which do not infect. Figure 5.2 is the reverse scenario where there are points that both infect and propagate. Here the latent failure rate is 0.2.

[1] Detailed in the tables in Chapter 6.

#### 5.1.1.3 Latent Failure Rate Measure for an Entire Program

The latent failure rate may be determined for the highest abstraction level, an entire program at the source level. The latent failure rate measure for an entire program $P$ is:

$$lfr_P = \min_{i \in PEC}[lfr_i] \tag{5.9}$$

This is the minimum latent failure rate estimate over all path equivalence classes; equation 5.9 depends on measurement scheme of the latent failure rate chosen at the path levels.

#### 5.1.1.4 The Hierarchical Method for Measuring the Latent Failure Rate

Equations 5.4 - 5.5 and 5.7 - 5.8 give a rough and conservative estimate of the latent failure rate for a path equivalence class. All equations presented thus far are a function of $\Delta$. Potentially a path equivalence class will have a location on it that yields a tiny ($< 0.01$) infection estimate. If this occurs, the latent failure rate produced by equations 5.7 - 5.9 will almost certainly be negative. So for paths of many unique locations, a less conservative scheme for determining the latent failure rate may be needed.

The *hierarchical method* presented in this section for finding a path's latent failure rate introduces, in general, a less conservative model than the prior schemes of measurement. The hierarchical model is a function of $\mathcal{F}$ and the hierarchical model does not produce negative latent failure rates. Equations 5.5 and 5.8 attempted to tighten up (make less conservative) the latent failure rate estimates by imposing a restriction that the infection used came from a location that precedes the location where the failure propagation estimate occurred. The hierarchical method takes this idea one step further by imposing the restriction that the infections used at a particular location come from actual faults at predecessor locations. The hierarchical method implementation of measuring the latent failure rate is not a function of $\Delta$. This is the main motivation for this measurement scheme. This model is strictly a function of semantic alternatives, both in their ability to infect and in the ability of the infections to propagate. This model alleviates concern that the measurement of the latent failure rate is a function of perturbation functions that simulate the effects of potential faults, since in general this set of faults is unknown.

The hierarchical method finds the latent failure rate for a path or subpath, not for a path equivalence class. The hierarchical method can, however, be adapted to a path equivalence class by making assumptions about the number of loop iterations.[2] In the hierarchical method, latent failure rates are a function of viral propagation estimates, whereas equations 5.4 - 5.5 and 5.7 - 5.8 use failure propagation estimates. And recall that viral propagation estimates are a function of the semantic alternatives of $\mathcal{F}$, not perturbation functions. Hence the main motivation mentioned in the previous paragraph is satisfied.

The hierarchical method suffers from the inability to properly handle path equivalence classes because it requires knowledge of every location executed and the order in which each location is executed; recall path equivalence classes are generalizations of potentially infinite sets of paths. Since the latent failure rate produced by the hierarchical method is specific to a path or subpath, it is necessary to have a unique viral propagation estimate and infection estimate for each location on the path or subpath as a function of the input space partition of the path.

Consider a program $P$ with 2 locations in sequence; $l_1; l_2$. The latent failure rate according to equation 5.7 is:

$$lfr_P = \min[I_{l_1} - (1 - \mathrm{F}_{l_1}),\ I_{l_1} - (1 - \mathrm{F}_{l_2}),\ I_{l_2} - (1 - \mathrm{F}_{l_1}),\ I_{l_2} - (1 - \mathrm{F}_{l_2})]$$

where $I_{l_x} = \min_z[I_{l_x,a_z}]$ and $\mathrm{F}_{l_x} = \min_q[\mathrm{F}_{l_x,a_q}]$. For a simple program like $P$, equations 5.4 - 5.5 and 5.7 - 5.8 may produce an accurate enough estimate of the true minimal failure probability relative to $\Delta$ or it may be negative. But for complex paths, equations 5.4 - 5.5 and 5.7 - 5.8 may produce too conservative of a latent failure rate.

This condition is evident in equations 5.7 and 5.4 which do not place constraints on the relationship between locations $i$ and $j$. Equation 5.5 and equation 5.8, which do place these constraints, produce a latent failure rate for a path equivalence class with an infection estimate from one location and potentially a failure propagation estimate from another. Although peculiar, this is more conservative. A scheme to avoid this in the latent failure for an entire program is:

$$lfr_P = \min_l[\min_k[I_{l,a_k}] - (1 - \min_z[\mathrm{F}_{l,a_z}])] \qquad (5.10)$$

Equation 5.10 is still peculiar since it forces an infection to occur at a location with no regard for whether that infection has any potential of occurring at that location whatsoever. In effect, these equations force the latent failure rate to be

[2] In the case of a trip set, there is the additional problem of a condition in a loop; an assumption will have to be made concerning which branch is taken on which iteration in order to generalize the hierarchical model to a trip set.

produced from estimates which force data states to be infected in a manner such that no fault could have ever caused that infection due to the use of a perturbation function.

The hierarchical method avoids this problem through viral propagation estimates. The hierarchical method requires that the infected data state that the viral propagation process uses be an infected data state that was created by the infection analysis process. There is the requirement that the original data states, $ds_{\mathcal{P}_1}$, used by the infection process are for the path that the hierarchical method is being applied to.

The hierarchical method algorithm is defined for a block of successive locations, where *successive* is interpreted dynamically rather than statically. Let the input to program $P$ be denoted by the data space $ds_{\mathcal{P}_1}$, and assume this data space is clean, so $\forall x \quad \mathcal{D}_{pred(1,1,x)}(x) = 0$, hence $\mathrm{V}_{1,1} = 0.0$. Note that $ds_{\mathcal{P}_1}$ might not represent the complete input set if location 1 is repeated. Denote the set of inputs that are represented by clean data states at data space $l$ on a particular iteration of location $l$ as $\{ds_{\mathcal{S}_l}$ is clean$\}$, i.e., the set of input points that have passed through the predecessor locations of location $l$ and have managed not to have their corresponding data states infected by the semantic alternatives that produced the minimum proportion of infections. The set of input points that have produced an infected data state in data space $ds_{\mathcal{S}_l}$ on a particular iteration of location $l$ is denoted as $\{ds_{\mathcal{S}_l}$ is infected$\}$. The set of inputs contained in $\{ds_{\mathcal{S}_1}$ is infected$\}$ represents the set of inputs that became infected by the semantic alternative that created the minimum infection estimate of location 1 on a particular iteration of location 1. The probability that $\{ds_{\mathcal{S}_1}$ is infected$\} \neq \emptyset$ is found by summing

$$\Pr[\{ds_{\mathcal{S}_1} \text{ is infected}\} \mid \{ds_{\mathcal{P}_1} \text{ is clean}\}] \approx \min_{a \in \mathcal{F}_1}\{I_{1,a}\}$$

and

$$\Pr[\{ds_{\mathcal{S}_1} \text{ is infected}\} \mid \{ds_{\mathcal{P}_1} \text{ is infected}\}] = \frac{\Pr[\{ds_{\mathcal{S}_1} \text{ is infected}\} \cap \{ds_{\mathcal{P}_1} \text{ is infected}\}]}{\Pr[\{ds_{\mathcal{P}_1} \text{ is infected}\}]}.$$

In this case, the second element summed is 0.0 on the first iteration of location 1 since $\mathrm{V}_{1,1} = 0.0$. Note that

$$\Pr[\{ds_{\mathcal{S}_x} \text{ is clean}\}] = 1 - \Pr[\{ds_{\mathcal{S}_x} \text{ is infected}\}] \tag{5.11}$$

The sets $\{ds_{\mathcal{P}_x}$ is clean$\}$ and $\{ds_{\mathcal{P}_x}$ is infected$\}$ are mutually exclusive for a particular iteration of location $x$, and their union is the sample space of input data states to the path. Returning to the example of successive locations $l_1; l_2$ which are not repeated,

$$\{ds_{\mathcal{S}_{l_1}} \text{ is infected}\} = (\{ds_{\mathcal{P}_{l_1}} \text{ is clean}\} \cup \{ds_{\mathcal{P}_{l_1}} \text{ is infected}\}) \cap \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}$$

$$= (\{ds_{\mathcal{P}_{l_1}} \text{ is clean}\} \cap \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}) \cup (\{ds_{\mathcal{P}_{l_1}} \text{ is infected}\} \cap \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\})$$

where $\{ds_{\mathcal{P}_{l_1}}$ is clean$\}\cap\{ds_{\mathcal{S}_{l_1}}$ is infected$\}$ and $\{ds_{\mathcal{P}_{l_1}}$ is infected$\}\cap\{ds_{\mathcal{S}_{l_1}}$ is infected$\}$ are mutually exclusive. So

$$\begin{aligned}
\Pr[\{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}] &= \Pr[(\{ds_{\mathcal{P}_{l_1}} \text{ is clean}\} \cap \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\})] + \\
&\quad \Pr[(\{ds_{\mathcal{P}_{l_1}} \text{ is infected}\} \cap \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\})] \\
&= \Pr[\{ds_{\mathcal{P}_{l_1}} \text{ is clean}\}] \cdot \Pr[\{ds_{\mathcal{S}_{l_1}} \text{ is infected}\} \mid \\
&\quad \{ds_{\mathcal{P}_{l_1}} \text{ is clean}\}] + \\
&\quad \Pr[\{ds_{\mathcal{P}_{l_1}} \text{ is infected}\}] \cdot \Pr[\{ds_{\mathcal{S}_{l_1}} \text{ is infected}\} \mid \\
&\quad \{ds_{\mathcal{P}_{l_1}} \text{ is infected}\}] \\
&\approx \Pr[\{ds_{\mathcal{P}_{l_1}} \text{ is clean}\}] \cdot \min_{a\in\mathcal{F}_{l_1}} \{I_{l_1,a}\} + \\
&\quad \Pr[\{ds_{\mathcal{P}_{l_1}} \text{ is infected}\}] \cdot \mathrm{V}_{l_1,1} \\
&= \min_{a\in\mathcal{F}_{l_1}} \{I_{l_1,a}\}
\end{aligned}$$

Similarly,

$$\Pr[\{ds_{\mathcal{S}_{l_2}} \text{ is infected}\} \mid \{ds_{\mathcal{S}_{l_1}} \text{ is clean}\}] \approx \min_{a\in\mathcal{F}_{l_2}} \{I_{l_2,a}\}$$

$$\Pr[\{ds_{\mathcal{S}_{l_2}} \text{ is infected}\} \mid \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}] = \frac{\Pr[\{ds_{\mathcal{S}_{l_2}} \text{ is infected}\} \cap \{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}]}{\Pr[\{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}]}$$

and

$$\begin{aligned}
lfr_{l_1;l_2} &= \Pr[\{ds_{\mathcal{S}_{l_2}} \text{ is infected}\}] \\
&\approx \Pr[\{ds_{\mathcal{S}_{l_1}} \text{ is clean}\}] \cdot \min_{a\in\mathcal{F}_{l_2}} \{I_{l_2,a}\} + \Pr[\{ds_{\mathcal{S}_{l_1}} \text{ is infected}\}] \cdot \mathrm{V}_{l_2,1}.
\end{aligned}$$

The equation for $lfr_{l_1;l_2}$ can be generalized to a path of $x$ dynamically executed locations in sequence. For any sequence of level 1 locations, the latent failure rate from location 1 through the predecessor locations (determined dynamically) of location $x$ on the i$^{th}$ iteration of location $x$ is defined recursively in terms of its predecessors by the following equations:

$$\Pr[\{ds_{\mathcal{S}_x} \text{ is infected}\} \mid \{ds_{\mathcal{P}_x} \text{ is clean}\}] \approx \min_{a\in\mathcal{F}_x} \{I_{x,a}\} \tag{5.12}$$

$$\Pr[\{ds_{\mathcal{S}_x} \text{ is clean}\} \mid \{ds_{\mathcal{P}_x} \text{ is infected}\}] \approx 1 - V_{x,i} \tag{5.13}$$

$$\Pr[\{ds_{\mathcal{S}_x} \text{ is infected}\} \mid \{ds_{\mathcal{P}_x} \text{ is infected}\}] =$$

$$\frac{\Pr[\{ds_{\mathcal{S}_x} \text{ is infected}\} \cap \{ds_{\mathcal{P}_x} \text{ is infected}\}]}{\Pr[\{ds_{\mathcal{P}_x} \text{ is infected}\}]} \tag{5.14}$$

$$\Pr[\{ds_{\mathcal{S}_x} \text{ is infected}\} \mid \{ds_{\mathcal{P}_x} \text{ is infected}\}] \approx V_{x,i} \tag{5.15}$$

$$\begin{aligned} lfr_x &= \Pr[\{ds_{\mathcal{S}_x} \text{ is infected}\}] \\ &\approx \Pr[\{ds_{\mathcal{P}_x} \text{ is clean}\}] \cdot \min_{a \in \mathcal{F}_x}\{I_{x,a}\} + \\ &\quad \Pr[\{ds_{\mathcal{P}_x} \text{ is infected}\}] \cdot V_{x,i} \end{aligned} \tag{5.16}$$

for the i$^{th}$ iteration of location $x$.

$$\Pr[\{ds_{\mathcal{S}_1} \text{ is infected}\}] \approx \min_{a \in \mathcal{F}_1}\{I_{1,a}\} \text{ since } \Pr[\{ds_{\mathcal{P}_1} \text{ is infected}\}] = 0 \tag{5.17}$$

Notice that equation 5.16 is the latent failure rate for a section of successive locations that ends on the i$^{th}$ iteration of location $x$; $x$ is the last location on either a path or subpath, not a path equivalence class $x$. So in equation 5.16, when $x$ is the exit node of a program, 5.16 represents the latent failure rate of a path. Equation 5.16 takes into account type I cancellation between locations with the term $V_{x,i}$.

**Observation 5.1:** *Equation 5.16 produces the most conservative latent failure rate estimate relative to $\mathcal{F}$ if and only if the viral propagation estimates are constant.* The following example shows a situation where chosing the minimum infection estimate produces a larger latent failure rate at a successor location than chosing the maximal infection estimate would. Suppose that $\{ds_{\mathcal{P}_l}$ is infected$\} = a$, $\{ds_{\mathcal{P}_l}$ is clean$\} = b$, and let $\mid a \mid = c$ for the i$^{th}$ execution of location $l$ on input $x$. Now let $\mid \{ds_{\mathcal{P}_{succ(l,i,x)}}$ is infected$\} \mid = d$, where $succ(l,i,x)$ is the successor location of location $l$ on the i$^{th}$ iteration of location $l$ on some input $x$, suppose that $d > c$, and suppose the semantic alternative causing the maximum infection estimate is used when determining how many elements of $b$ were moved into $\{ds_{\mathcal{P}_{succ(l,i,x)}}$ is infected$\}$. And let $\mid \{ds_{\mathcal{P}_{succ(l,i,x)}}$ is infected$\} \mid = e$ if the minimal infection estimate from a semantic alternative of $\mathcal{F}_l$ had been chosen when determining $\{ds\mathcal{P}_{succ(l,i,x)}$ is infected$\}$ instead of the maximal one. Thus $e < d$. Suppose also that the set created by the intersection of the set $e$ represents with the set $d$ represents is the empty set. Now suppose at location $succ(l, i, x)$ type I cancellation occurs strictly to those infected data states created by the input points in the intersection of the set $b$ represents with the set $d$ represents, however no type I cancellation occurs at location $succ(l, i, x)$ to the data states created by the input points in set of the intersection between the set $e$ represents with the set $b$ represents. If the effects of $I_{succ(l,i,x)}$ are ignored in determining $lfr_{succ(l,i,x)}$ using the hierarchical method, then

$$\left[\frac{\{ds\mathcal{P}_{succ(l,i,x)} \text{ is infected}\}}{\{ds\mathcal{P}_{succ(l,i,x)} \text{ is infected}\} \cup \{ds\mathcal{P}_{succ(l,i,x)} \text{ is clean}\}}\right]_{max} <$$

$$\left[\frac{\{ds\mathcal{P}_{succ(l,i,x)} \text{ is infected}\}}{\{ds\mathcal{P}_{succ(l,i,x)} \text{ is infected}\} \cup \{ds\mathcal{P}_{succ(l,i,x)} \text{ is clean}\}}\right]_{min},$$

where $[\cdot]_{max}$ represents some quantity $\cdot$ using the maximal infection estimate at location $l$, and where $[\cdot]_{min}$ represents some quantity $\cdot$ using the minimal infection estimate at location $l$. Unless the viral propagation rate is constant, a situation may occur where choosing the minimal infection estimate as shown in equation 5.27 may result in a greater latent failure rate than choosing a larger infection estimate for equation 5.27. To guarantee getting the lowest latent failure rate relative to $\mathcal{F}$, it is necessary to perform the combinatorially explosive action of unioning the set $\{ds\mathcal{P}_l$ is infected$\}$ for each semantic alternative at location $l$ with the set $\{ds\mathcal{P}_{succ(l,i,x)}$ is infected$\}$ for each semantic alternative at location $succ(l, i, x)$. As an example, suppose that at location $l$ three semantic alternatives were tried, producing the three infected sets: $\{a, b\}, \{c, d\}, \{a, b, c\}$ and suppose that the set of inputs still representing clean data states after executing location $l$ has the elements $\{f, g, h, k\}$. And suppose that after location $succ(l, i, x)$, where two semantic alternatives were tried, two new infected sets occur: $\{g\}$ and $\{h, k\}$ from $\{f, g, h, k\}$. To determine which semantic alternative to use as the semantic alternative at locations $l$ and $succ(l, i, x)$ for some $i$ and $x$, the sets $\{a, b, g\}, \{c, d, g\}, \{a, b, c, g\}, \{a, b, h, k\}, \{c, d, h, k\}$, and $\{a, b, c, h, k\}$ would need to be used during viral propagation analysis at location $succ(succ(l, i, x), i, x)$. This determines determine which semantic alternatives at locations $l$ and $succ(l, i, x)$ to claim as being the most conservative(in effect the semantic alternative that leaves the least elements in $\{ds\mathcal{P}_{succ(succ(l,i,x),i,x)}$ is infected$\}$. This situation becomes intractable after only a few locations.

The distinguishing characteristic between failure propagation and viral propagation is that viral propagation estimates are not a function of a perturbation function as are failure propagation estimates. Formally,

$$\begin{aligned}\{ds_{\mathcal{S}_1} \text{ is infected}\} = \{x \mid x \in \{ds_{\mathcal{P}_1} \text{ is clean}\} \ \wedge \ \forall a, a' \in \mathcal{F}_1 \\ \Pr[\mathit{infected}'(1, a, x) \mid \mathcal{B}_1(x) \neq \emptyset] < \Pr[\mathit{infected}'(1, a', x) \mid \mathcal{B}_1(x) \neq \emptyset]\}\end{aligned} \quad (5.18)$$

Equation 5.18 gives the definition for the first set of data states that starts driving equation 5.16 since $\{ds_{\mathcal{P}_1} \text{ is infected}\} = \emptyset$ on the first iteration. With equation 5.18 and the fact that $\overline{\{ds_{\mathcal{S}_1} \text{ is infected}\}} = \{ds_{\mathcal{S}_1} \text{ is clean}\}$, the hierarchical model is complete. Equation 5.18 can be generalized yielding:

$$\begin{aligned}\{ds_{\mathcal{S}_l} \text{ is infected}\} = \{x \mid x \in \{ds_{\mathcal{P}_l} \text{ is clean}\} \ \wedge \ \forall a, a' \in \mathcal{F}_l \\ \Pr[\mathit{infected}'(l, a, x) \mid \mathcal{B}_l(x) \neq \emptyset] < \Pr[\mathit{infected}'(l, a', x) \mid \mathcal{B}_l(x) \neq \emptyset]\} \cup \\ \{x \mid x \in \{ds_{\mathcal{P}_l} \text{ is infected}\} \ \wedge \ \mathcal{A}_l(x) \text{ is infected}\}\end{aligned} \quad (5.19)$$

for $l > 1$. Figure 5.3 gives an example of how the hierarchical method works. Initially, there is a state with ten clean data states and no infected data states. After executing location 1, which has a minimum infection estimate of 0.3, three of the data states have been lost into the right-most container for holding infected data states. After execution of location 2, another data state(really only half of a data state) is lost. After executing location 3, the low propagation rate effectively makes three of the infected data states become clean, hence they are moved back into the container for clean data states. This process continues through each succeeding location. The proportion of infected data states after execution of the exit node is the latent failure rate.

### 5.1.2 Software Faultprints

A *software faultprint* is a program characterization that parallels the notion of a human "fingerprint." It is believed (however not proven) that other than the code itself, a particular software faultprint of a program uniquely identifies the code it was created for, i.e., there is a one-to-one mapping from a program to a software faultprint relative to a particular input distribution. A software faultprint is a two part structure: the first half of a software faultprint is the information in Table 4.8; the second half of a software faultprint contains a ***dispersion histogram*** and an ***execution estimate*** for each path equivalence class as shown in Table 5.1.

The ***dispersion histogram*** reveals which input points are more likely to reveal

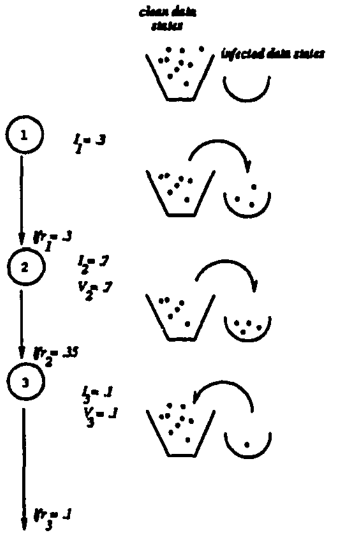


Fig. 5.3: Diagram showing latent failure rate determination by hierarchical method

faults. This is determined from previous empirical data, specifically the information produced during Process Propagation Analyzer. Process Propagation Analyzer, depending on the implementation of value distribution creation scheme, unfortunately discards the input point that created a particular data state. This input point is needed for creating the dispersion histogram. The dispersion histogram is the structure that records those input points which created data states that when perturbed caused pseudo-failure. The algorithm for building the histogram can be inserted directly into the algorithm of Process Propagation Analyzer. The abscissa-axis of the histogram represents bins of partitions of input points, and the ordinate-axis is the relative frequency of a point from that bin revealing a fault. An alternate scheme for representing the dispersion histogram is for the ordinate-axis to represent the number of pseudo-failures that points chosen from a bin have previously revealed.

The *execution estimate* for a path equivalence class $x$ is the proportion of inputs that cause a class $x$ to be executed. It is a function of the input distribution used.

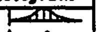
| *path equivalence class with corresponding regular expression* | *execution estimate* | *dispersion histogram* |
|---|---|---|
| 1 | $E_1$ |  |
| 2 | $E_2$ | |
| · | · | · |
| · | · | · |
| $n$ | $E_n$ | |

Table 5.1: Second half of a software faultprint

#### 5.1.2.1 Execution Estimate

The *execution estimate*, $E_x$, for path equivalence class $x$ of program $P'$ is the probability of executing path equivalence class $x$ with random inputs from a specific distribution. The distribution chosen is the distribution used to create the natural data states used for propagation and infection analysis. Let $T = \{t_1, ..., t_n\}$ and $TS = \{ts_1, ..., ts_k\}$. If $\mid T \mid = n$ and $TS = \emptyset$,

$$\sum_{i=1}^{n} E_{t_i} = 1.0. \tag{5.20}$$

An algorithm for estimating $E_x$ is:

1. set array counter_array to zeroes, where the size of counter_array = $\mid T \mid + \mid TS \mid$, with each counter_array element representing an element in $\mid T \mid + \mid TS \mid$,
2. instrument $P'$ with "write" statements that signal that a particular location was reached; this is similar to the algorithm for finding Method II value distributions,
3. execute $P'$ many times on input points,
4. feed the output to a pattern matcher with the regular expressions of the path equivalence classes; call the regular expression that matches the string $a$,
5. increment the corresponding counter_array[a].
6. divide each element of counter_array[a] by the number of times step 3 is performed.[3]

[3] The execution estimate is a point estimator which is the sample mean of the number of times the path equivalence class is executed, however to determine the confidence interval of the

The input distribution used is the same distribution as that for value distribution creation. If the run-time distribution is unknown, sampling is done from the uniform distribution. If $| TS | = k$ and $| T | = n$,

$$\sum_{i=1}^{n} E_{t_i} + \sum_{i=1}^{k} E_{ts_i} = 1.0 \qquad (5.21)$$

by definition of $TS$ and $T$.

**Execution estimates between versions**

It is desirable that execution estimates change minimally between versions. This is difficult to assure when the locations which uniquely define paths change. The necessity of execution estimates changing minimally in the software faultprint is not as crucial as for failure propagation or infection estimates in terms of computational costs. This is because $n$ program executions of $P''$ are made during determination of the execution estimates, whereas potentially $n$ partial executions of $P''$ are made for only one semantic alternative or one perturbed active variable (assuming Method III natural data state creation). Similarity of execution estimates between successive versions becomes a hard condition to assure when:

1. there is a control flow fault, or
2. there is missing code.

Many of the faults found late in the software development phase are "missing path" faults. This fault class may add additional path equivalence classes to the software faultprint; the new classes should have minimal execution estimates by the closely correct assumption. In fact, certain omitted location faults may not affect faultprint structure; it may occur that the set of path equivalence classes has elements already representing the altered paths that result as missing locations are added.

### 5.1.2.2 Dispersion Histogram

The *dispersion histogram* is the least important element in the software faultprint; it exists solely for efficiency in application models. It differs from the other

execution estimate, the confidence interval of the frequency distribution for 95% confidence can be found with $p \pm w$, where $w = 2 \cdot \sqrt{(p \cdot (1-p)/n)}$ and $p$ is the number of times that location was executed divided by the number of executions, $n$. For n = 10,000, accuracy is approximately ± 0.01 [17].

elements of the faultprint; it is not a statistic; not a measurable entity; it is a summary of empirical data. The intuition behind the dispersion histogram centers around the hypothesis:

**Hypothesis 5.1** *There exists some input point "a" that more likely exposes a fault than input point "b".*

Suppose it is decided to test a path equivalence class $i$ by uniformly picking $n$ points in $\phi_i$. If it were known that a particular subset of $\phi_i$, $\phi_{i_a}$, created data states that substantially disclosed more pseudo-failures during failure propagation analysis than did the remaining elements of $\phi_i$, then emphasizing sampling in $\phi_{i_a}$ during testing could be a better sampling strategy. In fact, using elements from $\phi_{i_a}$ may enable using fewer tests to achieve the same conclusions than if sampling is over all of $\phi_i$.

The development of a dispersion histogram involves building a histogram for each path equivalence class. For a path equivalence class $i$, $\phi_i$ is divided into bins along the abscissa-axis. It is assumed for simplicity in the discussion that each path equivalence class has one input value; $n$-dimension input variables are ignored. The dispersion histogram contains the same problem as Method II value distribution creation; storing $n$-dimensional values. The ordinate-axis of a dispersion histogram is either the probability density of a point from that bin revealing a fault or the number of pseudo-failures experienced from points in this bin. The first type is termed a *density dispersion histogram*, and the second type is termed a *failure dispersion histogram*. The density dispersion histogram's ordinate-axis is bounded between 0.0 and 1.0; the ordinate-axis is bounded by 0 and $\infty$(actually maxint) for the failure dispersion histogram. The abscissa-axis is bounded by the largest and smallest value in $\phi_i$. The density dispersion histogram may be derived from the failure dispersion histogram by dividing each bin's value by the product of the total number of perturbations per location and the number of locations on the path equivalence class. Note that the creation of the histogram for each path equivalence class does not require an oracle, and during tallying of the frequency count for a bin, it is assumed that each bin received equal sampling with respect to the width of the bin in Process Propagation Analyzer. The *width* of a bin is the cardinality of the set of input points which it represents, and the width of each bin is assumed to be the same size. Without equal sampling during histogram creation, the histogram is biased.

For simplicity, the issue is finessed of how to partition the input space of a path equivalence class by advocating placing adjacent elements along the abscissa-axis into bins. An example showing when this is a bad strategy follows: suppose there is a fault such that the program produces the correct output when the input is even, and fails when the input is odd. The "adjacent" binning strategy will not take the even-odd discontinuity into account, and it will produce a histogram with bins of similar frequency counts. It is not the intent of this thesis to explore the continuity decision. It however is a question that should be addressed if at all possible to maximize the information available from failure propagation analysis.[4]

Let $h_i$ denote the failure dispersion histogram for the i$^{th}$ path equivalence class. And let $b$ represent the number of bins and let $z$ represent the width of a bin.[5] The greater the number of bins, the greater the amount of information given; the fewer the number of bins, the lesser the amount of information contained in the histogram. Too few bins is effectively the same as not having the histogram.

Building the dispersion histogram need not be an isolated task. If it is known for each data state used during failure propagation analysis whether a pseudo-failure occurred and which input created that data state, then the frequency count for the bin in $h_i$ corresponding to that input point can be incremented. The current scheme by which failure propagation estimates are determined is to sample from the value distributions available for a location and execute until termination. The problem is that the particular input point that yielded that sampled data state is not known; it was not stored in either the value distribution creation algorithms of Method I or II. An algorithm which determines failure propagation estimates and builds the failure dispersion histogram simultaneously is:

1. Use value distribution creation Method III, therefore the input value corresponding to a particular data state can be easily stored,

2. Sample a point from $\phi$, determine the corresponding path equivalence class that it corresponds to, $i$,

3. For each location on the path equivalence class, perform failure propagation analysis on this element of $\phi_i$ for each active variable,

4. Each time a pseudo-failure occurs for this element of $\phi_i$ at a perturbed location on path equivalence class $i$ during failure propagation analysis, increment the counter representing the bin this input is from; for example, if there are 100 locations and 5 active variables per location, then potentially a pseudo-failure may occur 500 times for just one particular input point, so the frequency count of the bin representing the input point would be increased by 500,

[4] Hints for building both discrete and continuous histograms are found in [17].

[5] Assume that there is a bin for each element of $\phi_i$, thus no outliers, i.e., each point corresponds to a specific bin.

5. Perform the previous 3 steps for $n$ elements of $\phi_i$, until there is a histogram for each path equivalence class $i$; the larger $n$, the more information in $h_i$.

It is not shown in this thesis that the results do not change between successive versions of a program for the dispersion histograms.[6]

### 5.1.2.3 Impact of Discovering Faults on the Software Faultprint

Measurement of a particular fault's impact on a program is difficult. Different fault classes will impact the faultprint structure and faultprint contents differently. The impact is a function of where in the program the fault occurs. The *faultprint structure* refers to the path equivalence classes and locations; *faultprint contents* refers to the internal information the software faultprint contains. This information includes the path equivalence class regular expressions, execution estimates, failure propagation estimates, infection estimates, and the dispersion histogram. Omitted location faults may potentially change the structure of the faultprint once discovered, however under the closely correct assumption such faults should be of small fault size since current testing is not finding them. Different types of faults will affect the faultprint differently; thus a distinction between computational faults and control flow faults is made. *Computational faults* which have no effect on control flow do not affect the faultprint structure however execution estimates may change. A location $l$ contains a computational fault if the output location $z$ is data dependent on location $l$. *Control flow faults* may change the structure of the software faultprint and thus the execution estimates. A location $l$ contains a control flow fault if the output location $z$ is control dependent on location $l$. Programs with control flow faults or computational faults of small size can be analyzed with propagation and infection analysis without substantial changes to the faultprint between successive versions. There are faults which are classified as both computational fault and control fault. Such faults when removed will have an unpredictable affect on the software faultprint, which will depend on the program, the fault, the input distribution, and the location within the program.

## 5.2 Auxiliary Application Processes

This section describes four additional auxiliary processes. Process Dispersion Histogram Producer produces dispersion histograms for path equivalence classes.

[6] Chapter 6 shows failure propagation estimate similarities between versions, this observation will hold for dispersion histograms as well.

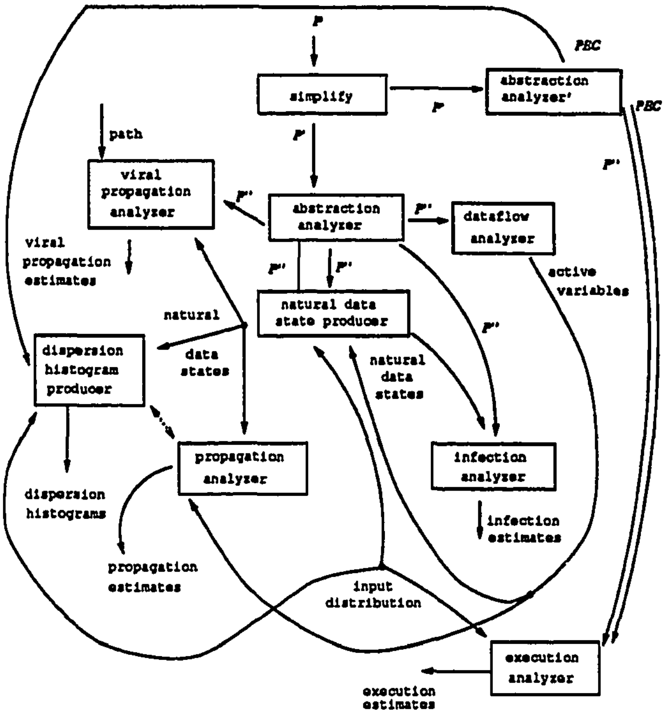


Fig. 5.4: Processes for applications of PIA

Process Execution Analyzer produces execution estimates for path equivalence classes. Process Abstraction Analyzer' identifies the regular expressions according to the definitions of the path equivalence classes. Process Viral Propagation Analyzer produces viral propagation estimates for specific paths. Figure 5.4 shows the interaction of these processes with the processes for PIA.[7]

### 5.2.1 Process Abstraction Analyzer'

This process performs the exact task as Process Abstraction Analyzer, however it performs the additional task of producing the path equivalence classes. Input to Process Abstraction Analyzer' is $P'$ and the definitions for the path equivalence

[7] The dotted line between processes means the processes may be combined into one, done separately, or concurrently.

classes.

### 5.2.2 Process Dispersion Histogram Producer

Process Dispersion Histogram Producer performs the algorithm given in Section 5.1.2.2; Process Dispersion Histogram Producer takes $P''$ as input, the regular expressions of the path equivalence classes from Process Abstraction Analyzer', and either the value distributions from Process Natural Data State Producer or $\phi$. If the value distributions are supplied, there must be a way to backtrack to find the input point that created a particular data state. Output from this process is a dispersion histogram for each element in $PEC$.

### 5.2.3 Process Execution Analyzer

Process Execution Analyzer performs the algorithm in Section 5.1.2.1; Process Execution Analyzer has as input $P''$, the regular expressions of the path equivalence classes, and $\phi$. This process produces an estimate of the frequency that a particular path equivalence class is executed for each path equivalence class.

### 5.2.4 Process Viral Propagation Analyzer

This process performs the algorithm for determining viral propagation estimates. Viral propagation estimates are a function of a path, so input to Process Viral Propagation Analyzer is $P''$ and the set of input points specific to the path. Output from this process is a viral propagation estimate, $V_{x,i}$, for the $i^{th}$ iteration of each location $x$ for some path.

## 5.3 Applications

In Section 5.3, six applications of propagation and infection analysis are enumerated and explained: Section 5.3.1 describes the model for probable correctness; Section 5.3.2 describes software reliability; Section 5.3.3 describes ultra-reliability; Section 5.3.4 discusses a software testing strategy; Section 5.3.5 explains debugging; and Section 5.3.6 describes a testing complexity metric.

### 5.3.1 Probable Correctness

It is important to relate the ***specification*** to reliability, since a specification alone can determine correctness for any input/output pair. The type of correctness defined in Chapter 2 is *functional correctness.* Correctness may also be deduced using valid rules of inference applied to axioms. This is termed *axiomatic correctness*, where the semantics of each language construct are used in constructing the proof. Hoare [5] terms the initial predicate of the construct or program a *precondition*, the final predicate a *postcondition*, and uses the notation $P\ \{Q\}\ R$ to say that if $P$ is originally true, that after execution of construct $Q$, $R$ is also true. For instance, given the statement `a := a - 1` with the initial assertion $\{a > b, b = 5\}$, the assertion $a \geq 5$ and $b = 5$ is made afterwards. Typically assertions are made after particular variables are assigned values at reachable locations in the program. The assertions usually do not specify values for the variables, but assert relationships among variables.

This method has several deficiencies as noted by Hoare [5]. First, if the programming language allows side-effects, the proof system must show their absence in each execution. Secondly, termination must be shown in any proof scheme, and in order to prove this, knowledge of implementation-dependent features may be required. Hence it is more common to discuss *partial* correctness, which says that "if the program halts, it is correct". Another problem is determining whether the proof is correct or not, which may be difficult.

Since in general proofs of correctness are unavailable, Hamlet [4, 22] proposes the notion of "probable correctness." *Probable correctness* [4, 22] attempts to predict from a successful test of the program whether there are no faults in the program. This is different than software reliability, which attempts to predict the probability or frequency with which the program will fail, or the mean time until the next failure, or even the number of remaining faults[10, 1, 15, 20]. Probable correctness is in some some sense the degenerate case of software reliability, because it is the probability of non failures, a mean time until the next failure of $\infty$, and zero remaining faults.

Valiant [23] produces a model that can be used in probable correctness. Given a space with $X$ types of objects, sample $n$ objects independently, and suppose that not all of the object types appear. If there exists a probability $p$, it can be used to determine a value $n$ that guarantees that the probability of these so far unseen objects occupying less than $p$ of the total space is at least $1 - p$. [23]. Valiant [23] produces a model that can be used for determining probable correctness. Given a space with $X$ types of objects, sample $n$ objects independently, and suppose that not all of the object types appear. If there exists a probability $p$, it can be used to determine a value $n$ that guarantees that the probability of these so far unseen objects occupying less than $p$ of the total space is at least $1-p$. [23]. Valiant's argument [23, 22] can be summarized as follows. Assume a urn has two types of balls: black and white. The number of balls in the urn may or may not be infinite, however sampling from the urn is with replacement, so effectively the number of balls is infinite. Let $p$ represent the unknown proportion of black balls in the urn, and let the discrete random variable $X$ represent the number of black balls that will be drawn with replacement in a sample size of $n$. Then the associated binomial probability density function of $X$ is:

$$\Pr[X = x] = \binom{n}{x} p^x (1-p)^{n-x}$$

where the possible values of $X$ are $x = 0, 1, 2, ..., n$. If drawing a white ball is viewed as a program success and drawing a black ball as a program success, the probability of no failures occurring in $n$ executions is:

$$\Pr[X = 0] = (1-p)^n$$

and so the probability of at least one failure occurring in $n$ executions is:

$$\Pr[X > 0] = 1 - (1-p)^n \tag{5.22}$$

Drawing from the urn can be viewed as a hypothesis test as follows: let the null hypothesis $H_0$ be that the proportion of black balls in the urn is $= 0$, and let the alternative hypothesis $H_a$ be that the proportion of black balls in the urn is $> 0$. The test is to draw $n$ balls; if no black balls are drawn, the conclusion is that $H_0$ is true; if one or more black balls is drawn, the conclusion is that $H_0$ is false.

| $H_0$ is true | $H_0$ is false | |
|---|---|---|
| o.k. | type II error | conclude $H_0$ is true |
| type I error | o.k. | conclude $H_0$ is false |

The type I error is the conditional probability:

$$\alpha = \Pr[\text{type I error}] = \Pr[\text{reject } H_0 \mid H_0 \text{ is true}] = 0.$$

The type II error (the "serious" error) is the conditional probability:

$$\begin{aligned} \beta &= \Pr[\text{type II error}] = \Pr[\text{accept } H_0 \mid H_0 \text{ is false}]. \\ &= (1-p)^n \end{aligned}$$

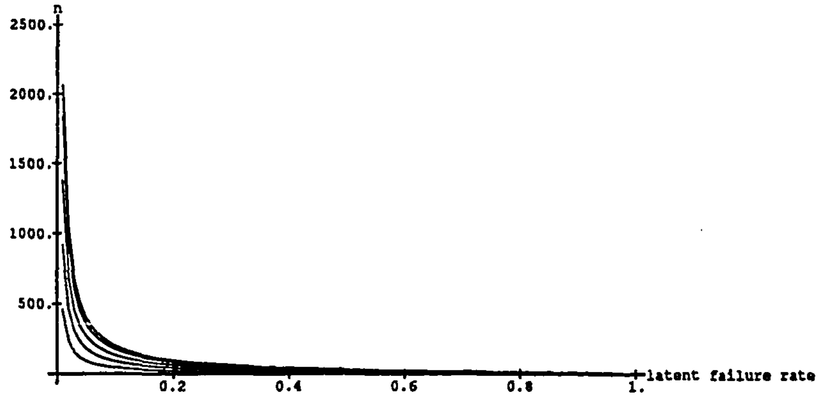


Fig. 5.5: $n$ vs. Latent failure rate for various $\alpha$s

Then $1-\beta = 1-(1-p)^n$ is the probability of no type II error, i.e., $1-\beta$ is the confidence that there are no black balls in the urn. Figure 5.5[8] is a graph of the various confidences provided for two parameters: $n$ and $p$.

#### 5.3.1.1 Applying Dispersion Histograms and Latent Failure Rates to Probable Correctness

From equation 5.22 and [25],

$$1-[1-lfr_i]^{N_i} = \alpha_i \tag{5.23}$$

is defined as the probability of at least one failure while executing path equivalence class $i$ in $N_i$ executions of path equivalence class $i$. Then

$$N_i = \lceil \frac{\ln(1-\alpha_i)}{\ln(1-lfr_i)} \rceil \tag{5.24}$$

is the number of points that execute path equivalence class $i$ which must be chosen for a $\alpha_i$ confidence that path equivalence class $i$ is correct.

It is assumed in equation 5.23 that the points are chosen uniformly from $\phi_i$,

[8] Confidences in order from point (0,0): 0.01, 0.0001, 0.000001, 0.00000001, 0.000000001.

and that the distribution of $N_i$ is the distribution used for finding $lfr_i$.[9] From the dispersion histogram, knowledge exists as to which points in $\phi_i$ that are more likely to reveal a fault. So if these points exist and emphasis is placed on sampling from them, this should allow for a decreased sample size than $N_i$ with equivalent confidence in the assertion that path equivalence class $i$ is correct.

Let $p$ represent the probability of some event occurring on the first trial. The probability of the event occurring on the first or second attempt is $p + p(1 - p)$. The probability of it occurring on the first, second, or third attempts is $p + p(1 - p) + p(1 - p)^2$ and so on. Now reinterpret $p$ to represent the probability of an event occurring from one of the lower frequency count bins in the histogram, and let $w$ represent the probability of the event occurring from the highest frequency count bin. A value of $z$ is desired such that

$$(\underbrace{\underbrace{\underbrace{p}_{1}+p(1-p)}_{2}+p(1-p)^2.....}_{z}) = w.$$

Let $m$ represent the bin in failure dispersion histogram $h_i$ denoted by $h_i[m]$ with the greatest frequency count[10], and let $f$ be the subset of $\phi_i$ that bin $m$ represents; $f$ is the set of input points believed to most likely reveal a fault. Let $n = \sum_{y=1}^{b} h_i[y]$, let $a_k$ denote the number of points that if chosen from a bin $k$, where $k \neq m$, will equal the probability of finding a fault with one point from $m$[11], let $c_k$ represent the number of points that would have been chosen from bin $k$ but are now substituted by points from bin $m$, and let $\{p_1, p_2, ...\}$ be a set of temporary variables used for holding probabilities needed for determining $a_k$. So for any bin $k$, the probability of the first randomly chosen point from that bin revealing a fault is:

$$p_1 = \frac{h_i[k]}{n}. \tag{5.25}$$

The probability that the j$^{th}$ randomly chosen point from the k$^{th}$ bin will reveal a fault is now defined. This is the probability of the event occurring ($p_1$) multiplied by the probability of it not occurring on the previous $j - 1$ iterations. This

[9] Another implicit assumption is $N_i \gg b$, otherwise the following algorithm can not be applied.

[10] If there are $x$ bins that have the same maximum count, consider them as one bin with $f$ equal to the union of the subsets of $\phi_i$ that those $x$ bins represent; and let the width of the new bin equal $zx$; however for simplicity assume there is one maximum bin.

[11] $a_k$ should be smallest value that satisfies the inequality 5.28.

probability is:

$$p_j = \frac{h_i[k]}{n}(1 - \frac{h_i[k]}{n})^{j-1} \qquad \text{for j} \geq 2 \tag{5.26}$$

Then the probability that the event will occur on either the first or the second or the third time, etc., is the sum of these $p_j$s. Equation 5.27 gives the number of draws, $a_k$, from the k$^{th}$ bin that are probabilistically equivalent to one draw of revealing a fault from bin $m$. $a_k$ should be the smallest value such that equation 5.27 is true.

$$\frac{h_i[m]}{n} \leq \sum_{j=1}^{a_k} p_j \tag{5.27}$$

If $a_k$ is small, then potentially no savings in the number of points needed from bin $k$ can be achieved. However if $a_k$ is large, then the savings from bin $k$ is denoted by $c_k$. Safety is provided by at least sampling one point from bin $k$ regardless of $a_k$, so one point is subtracted from the gross savings (assuming there is a savings).

$$c_k = \begin{cases} \lfloor(1 - \frac{1}{a_k}) \cdot (\frac{N_i}{b})\rfloor & \text{if } \lfloor(1 - \frac{1}{a_k}) \cdot (\frac{N_i}{b})\rfloor = 0 \\ \lfloor(1 - \frac{1}{a_k}) \cdot (\frac{N_i}{b})\rfloor - 1 & \text{otherwise} \end{cases} \tag{5.28}$$

The total savings is just the sum of the $c_k$s over all bins except bin $m$.

$$\gamma = [\sum_{\{k|1 \leq k \leq b, k \neq m\}} c_k] \tag{5.29}$$

So the new sample size, $\hat{N}_i$, is:

$$\hat{N}_i = \lceil \frac{\ln(1 - \alpha_i)}{\ln(1 - lfr_i)} \rceil - \gamma. \tag{5.30}$$

$\hat{N}_i$ represents the new estimate of the number of points needed for confidence $\alpha_i$ when sampling is done across $h_i$. $\gamma$ represents the savings in the number of samples necessary for $\alpha_i$. When choosing where in $h_i$ to sample the $\hat{N}_i$ points, the following scheme is used:

1. Sample one point from every bin whose frequency count is $< h_i[m]$; denote the number of bins meeting this requirement by $\mu$,

2. Sample $\hat{N}_i - \mu$ points from bin $m$.

Sampling from those bins with relative frequencies are less than $\frac{h_i[m]}{n}$ is a precautionary measure to assure that at least one point is executed from each bin regardless of the bin's relative frequency. This is intuitive. It is possible that for an arbitrary fault, the information suggested in the histogram does not hold; if true and only bin $m$ is sampled from, then false conclusions could result. Equation 5.28 takes into account that $\mu$ points are chosen from bins with lesser relative frequencies than $\frac{h_i[m]}{n}$. Notice that when $\lfloor(1-\frac{1}{a_k})\cdot(\frac{N_i}{b})\rfloor = 0$, no elements from bin $k$ are ignored. But when $\lfloor(1-\frac{1}{a_k})\cdot(\frac{N_i}{b})\rfloor > 0$, $\lfloor(1-\frac{1}{a_k})\cdot(\frac{N_i}{b})\rfloor$ points can mathematically be ignored, however a point from each of the $\mu$ bins is tried, hence $\lfloor(1-\frac{1}{a_k})\cdot(\frac{N_i}{b})\rfloor$ is decreased by one.

The algorithm for implementing equation 5.29 is:

```
$i := 1$
$p := h_i[k]/n$
while $p \leq h_i[m]/n$ do
    $p := p + h_i[k]/n(1 - h_i[k]/n)^{i-1}$
    $i := i + 1$
$\gamma_k := ((1 - (1/i)) \cdot (N_i/b)) - 1$
```

As an example, let $h_i$ be a 4 bin density dispersion histogram with relative frequencies $h_i[1] = 0.05, h_i[2] = 0.05, h_i[3] = 0.6, h_i[4] = 0.3$, $z = 1000$, and $N = 400$. Sample 100 points from bin 3. From bins 1, 2, and 4 sample one point each. Then from bin 3 pick 46 more points, for a total sampling of 149 points. A savings of 251 points to achieve an equivalent confidence that the program is correct is substantial since the points used in determining the correctness require an oracle. Input points requiring an oracle are expensive and a reduction in the number is welcome.

A minor problem occurs if $N_i < b$.[12] Since $\hat{N}_i \leq N_i$ is desired, if $N_i < b$, then $\hat{N}_i < b$. One solution is to consolidate bins and decrease $b$ until $N_i \geq b$, however this decreases the information contained in the bins and this is a bad alternative. The best sampling strategy appears to be:

- Sample one point from each of the $N_i$ bins that have the largest density

Intuitively, it is safer to ignore the $\gamma$ savings in points and attempt to sample over as many bins as possible.

[12] This is probably an unrealistic case.

### 5.3.2 Software Reliability

An overview of software reliability terminology and models is found in [10, 1, 15, 20]. Most of these are black-box models. This section introduces a white-box method of applying propagation and infection analysis to software reliability.

#### 5.3.2.1 Applying Latent Failure Rates in a Software Reliability Model

The software reliability equations presented in this section are conservative. They are designed to underestimate the true reliability with high assurance that the estimate is conservative. Equation 5.31 sums the execution probabilities of all path equivalence classes whose latent failure rates are substantially larger than the observed failure rate. So this equation and the remaining equations for software reliability require an oracle. With this, the reliability estimate for program $P$ using the software faultprint is:

$$R_P = \sum_{\{x|(x\in\{PEC\})\wedge(E_x\cdot lfr_x \gg \lambda_P)\}} E_x \tag{5.31}$$

Equation 5.31 is a direct result of Hypothesis 5.2:

Hypothesis 5.2: *If there exists an $E_t \cdot lfr_t \ll \lambda_P$ or $E_t \cdot lfr_t \approx \lambda_P$, then the path equivalence class $t$ may contain the fault causing $\lambda_P$; however, if the $E_t \cdot lfr_t \gg \lambda_P$, then path equivalence class $t$ is almost certainly not responsible for $\lambda_P$.*

Equation 5.31 is conservative because it excludes path equivalence classes with low latent failure rates even though these path equivalence classes may be correct. An alternative to this is to take the information produced in the probable correctness model. Let $\alpha_x$ represent the confidence that path equivalence class $x$ is correct, i.e., path equivalence class $x$ has been tested a particular number of times and no failures were observed. Then

$$R_P = \sum_{\{x\in PEC\}} E_x \tag{5.32}$$

with a confidence in the reliability estimate of at least $\min_x\{\alpha_x\}$. Equation 5.32 has the possibility of producing an assertion of 100% reliability with a tiny confidence in this assertion. An alternative to equation 5.32 that ignores path equivalence classes which have not shown substantial confidence in the probable correctness model is:

$$R_P = \sum_{\{x \in PEC\} \wedge \alpha_x \approx 1.0} E_x \tag{5.33}$$

Equation 5.33 limits the path equivalence classes that are included in the summation by requiring that the confidence in the assertion that path equivalence class $x$ is correct be high(by forcing $\alpha_x \approx 1.0$). Equation 5.33 produces the reverse problem of equation 5.32; a high confidence in a potentially low reliability estimate.

None of these software reliability equations produced are considered as "good." They are included as introductory schemes for applying propagation and infection analysis's results to an important field.

### 5.3.3 Ultra-Reliability

A program $P$ is said to be *ultra-reliable* if it satisfies:[13]

$$\lambda_P \approx 10^{-x}.$$

Equations 5.31 - 5.33 do not produce as accurate a value as $(1 - 10^{-x})$ for large $x$, however propagation and infection analysis should not be overlooked as a tool in ultra-reliability. A scheme for applying propagation and infection analysis to ultra-reliability validation is to take the information in Table 4.8 and isolate those locations with tiny infection and failure propagation estimates, particularly those which are less than $10^{-6}$. These estimates indicate locations which can hide faults that conventional testing may not isolate. Execution estimates at the location abstraction level also may be used during ultra-reliability validation in a similar manner. With this information, these locations can be formally proven or tested in isolation.

### 5.3.4 Software Testing

*Dynamic testing* plays a significant role in many software reliability models, because without testing, a program's failure rate could not be determined. Branch, statement, path analysis, error-based, boundary-value, domain, and mutation

[13] The units are quantified as $10^{-x}$ as the probability of failure per 10 hours, thus if the program executes $10^6$ times per hour, then to be certified as ultra-reliable, $P$ must be shown to have a failure probability of $10^{-x-5}$.

testing are examples of software testing strategies; all have limitations. *Statement testing* is achieved when every statement is executed at least once; *branch testing* requires that each branch be executed at least once. *Mutation testing* [21] is a strategy which evaluates the test cases, by taking a program $P$ and producing $n$ versions(*mutants*) of $P$, $[p_1, p_2, ...p_n]$, that are syntactically different from $P$. If the test cases differentiate the mutants from $P$, then it is assumed that if the actual program works with those test cases, the program is good. Mutation testing assumes the "competent programmer hypothesis" which states that a competent programmer produces code that is close to being correct; where close means with a few syntax changes it should be correct. It also assumes that faults that interact can be caught with test data that reveals single faults, i.e., fault coupling is ignored. *Error-based testing* attempts to define certain classes of errors and the subdomain of the input space which should reveal any error of that class if that error type exists in the program [16]. Morell [14] proves properties about error-based strategies concerning certain errors that can and cannot be eliminated using error-based testing. Since error-based testing restricts the class of computable functions, it is limited as well. *Fault-seeding* is an error-based technique used to estimate both the number of faults remaining as well as their type. Faults are seeded and the "seeded" version is then run. Based upon the number of faults discovered, an estimate of the number of remaining faults is made [11]. The drawback is that if the seeded faults are not representative of the inherent faults, the estimate is invalid. *Domain testing* is another error-based testing strategy which partitions the input domain according to the program's paths. Each partition is termed a path domain, and faults which cause an input to be in the wrong path domain is a domain error. Domain testing attempts to discover faults by using test data that limits the range of undetected faults [2].

*Special-value testing* selects test data based upon special properties of the function being computed. For instance, for trigonometric functions, inputs such as 0 and $2\pi$ would be selected. *Path analysis* [8] attempts to select points which cause certain paths to be executed; however it is an unsolvable problem to determine for an arbitrary location in a program if there exists an input point which reaches that location. To see this, consider the exit point of the program as the location, and it is desired to know if the exit location can be reached by any input point. If the feasibility problem can be solved , then so can the *Halting* problem. But the *Halting* problem is not solvable, hence neither is the feasibility problem. Path analysis is also intractable whenever a program has an indefinite loop due to infinitely many paths.

Statement testing is a widely used measure of test coverage [6], however a very important class of errors cannot be detected, namely incorrect flow of control [9]

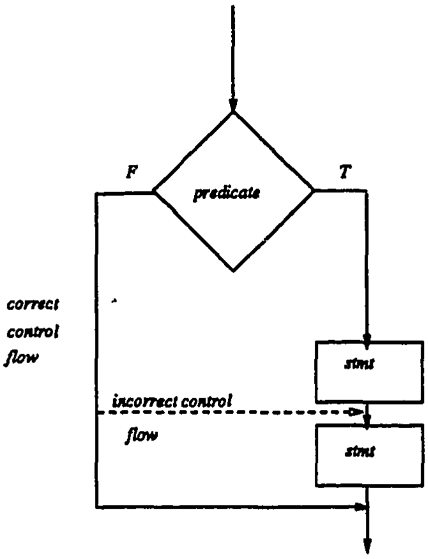


Fig. 5.6: Incorrect control flow

as shown in Figure 5.6. Statement testing is subsumed by branch testing; it may leave branches untested. Branch testing would catch this error that statement testing missed. Mutation testing is impractical because of the infinite number of alternatives possible, and only a handful can be used in trying to evaluate the coverage of the test cases. Criticism has arisen as to whether the mutant programs actually correspond to reasonable faults. But what is a reasonable fault? Must it be a fault that a human would interject, not a machine, or would it be better to define a mutated fault as equivalent to a "real" fault if the probability of catching the mutated fault is the same as the probability of catching the real fault. In [25], the notion of *stratified fault seeding* is presented, where faults may be seeded with varying degrees on the reliability impact, thus the argument as to whether these faults are realistic is unnecessary.

So which testing strategy is best? It seems clear that testing with only one strategy probably does not produce results as good as using a combination of strategies that take into account peculiarities of the code and the specification [7]. Testing theories consider ideal testing scenarios versus what is physically possible. Howden [8] defines a *reliable* testing strategy as one which reveals a fault whenever one exists, which in general is impossible, due to the large number(often infinite) of inputs required to show correctness(100% reliability). As pointed out by Huang

[9], for a program with two 32-bit integers, there are $2^{64}$ possible combinations, and if each combination requires a millisecond, it takes 50 billion years to test all combinations. Goodenough and Gerhart [3] define an *ideal* test as one which succeeds only when the program is correct. They define a data selection criterion C as *reliable* if it is necessary to test only one element of a particular input partition, and *valid* if the criterion does not forbid the picking of a test case that produces failure. For them, a reliable test set is one in which every element in a partition either fails or succeeds.

A comprehensive synopsis on software testing techniques is provided in [13]. Further discussion of existent testing techniques is curtailed. Section 5.3.4.1 shows a scheme for applying propagation and infection analysis to software testing.

#### 5.3.4.1 Applying Software Faultprints to Software Testing

The execution estimates, infection estimates, and failure propagation estimates from the software faultprint can guide software testing. Although nothing absolute can be said about the degree of testing required from propagation and infection analysis's output (other than that stated in equation 5.23,[14]) it is clear that that for two distinct path equivalence classes $x, y$, if

$$E_x \cdot lfr_x \ll E_y \cdot lfr_y, \tag{5.34}$$

then path equivalence class $x$ should receive either more testing than path equivalence class $y$ or possibly testing from $h_x[m]$. If $lfr_x \leq lfr_y$ or $lfr_x \approx lfr_y$ then testing is a function of of $E_x$ and $E_y$.[15]

If testing is at the location abstraction level, then for two distinct locations $i$ and $j$, if

$$E_i \cdot lfr_i \ll E_j \cdot lfr_j,$$

where $E_i$ is the proportion of times location $i$ is reached per $n$ program executions, then location $i$ should either receive more testing than location $j$ or possibly verification. Once again, for similar values of $lfr_i$ and $lfr_j$, determine execution estimates at the location abstraction level to determine if either location is executed more frequently. This is a reasonable scheme to determine if a location

[14] See Section 5.3.6.

[15] The execution probability has been separated out from the latent failure rate in the application models as often as possible because input distributions are volatile and may change and it is desired that the results are not held "hostage" by such occurrences. In this case, more testing is warranted of path equivalence class $x$ if $E_x > E_y$ since their latent failure rates are approximately the same and $E_x$ is more frequently traversed.

should receive special consideration. If $\zeta_x$ is large, then location $x$ may require a more formal approach than testing; possibly a correctness proof.

### 5.3.5 Debugging

Debugging has one goal: *to find and correct faults.* In general, there are three methods to debugging: backtracking, brute force, and cause elimination [18]. Brute force is commonly used; it takes the attitude of letting the computer do the work by either injecting "write" statements or invoking run-time traces [18]. Cause elimination is performed by organizing data that is related to the error occurrence. Then a cause hypothesis is devised, and the data is used to either prove or disprove the hypothesis [18]. Backtracking starts at the location where the trouble began, and the code is analyzed backwards through the sequence of executed locations. For large programs this may be unbearable.

Automatic debugging tools are available. These include debugging compilers, automatic test case generators, and dynamic debugging tracers [18]. All of these methods either require a specification or someone who can watch a trace or memory dump to see what is happening.

Debugging is a two step process: locating the fault first, and then correcting it. Debugging using propagation and infection analysis aids more in the first of these two steps than in the second. A person familiar with the system will still have to perform the second step; propagation and infection analysis has no knowledge about the specification, however propagation and infection analysis gives an indication of what fault class might be causing the failure rate and at what location. Propagation and infection analysis's usefulness is in quickly narrowing the number of locations that will have to be checked manually. To perform debugging using propagation and infection analysis, no knowledge of the code or specification is required. For debugging to succeed, ***it is assumed that the faults are single faults, not distributed faults.*** To perform debugging, four parameters are necessary:

1. the failure rate of the program $P$, $\lambda_P$,

2. the input distribution, $\phi$, causing $\lambda_P$,

3. the complete results from propagation and infection analysis shown in Table 4.8.

4. the execution estimate of each location $z$, $E_z$, relative to the input distribution.

The first of two algorithms to perform debugging using propagation and infection analysis is:

1. to determine if a location $z$ is a *suspect*, where a suspect is defined as a location that is thought to be contributing to the current failure rate, take each $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair, get the product, then multiply this product by $E_z$, and compare this new product to $\lambda_P$. If

$$\exists k\ \exists j\ E_z \cdot I_{z,a_k} \cdot \mathrm{F}_{z,a_j} \gg \lambda_P, \tag{5.35}$$

then an incorrect active variable $a_j$ and semantic alternative $a_k$ at location $z$ is not further considered as potentially causing $\lambda_P$; So in effect, a matrix of values occurs when the cross product of the infection estimates and the failure propagation estimates is performed. Ignore the pairs satisfying inequality 5.3; they are not suspects,

2. If any $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair such that $\exists k\ \exists j\ E_z \cdot I_{z,a_k} \cdot \mathrm{F}_{z,a_j} \approx \lambda_P$ is considered suspicious, thus location $z$ is a suspect. If every $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair at location $z$ is not suspicious, then location $z$ is not a suspect.

3. those $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pairs where $\exists k\ \exists j\ E_z \cdot I_{z,a_k} \cdot \mathrm{F}_{z,a_j} \ll \lambda_P$ can not be ignored; by the single fault assumption, there may be several $I_{z,a_k} \cdot \mathrm{F}_{z,a_j}$ summing up to $\approx \lambda_P$. The set of combinations where

$$\exists k_1\ \exists j_1\ E_l \cdot I_{l,a_{k_1}} \cdot \mathrm{F}_{l,a_{j_1}} + \ldots + \exists k_n\ \exists j_n\ E_p \cdot I_{p,a_{k_n}} \cdot \mathrm{F}_{p,a_{j_n}} \approx \lambda_P \tag{5.36}$$

is true therefore forces locations $l, \ldots, p$ to be considered as suspects.

4. the set of locations having at least one $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair considered as suspicious is the set of suspects; this is the set from which either static or dynamic analysis of the code can begin.

These rules ignore the case where $\exists k\ \exists j\ E_z \cdot I_{z,a_k} \cdot \mathrm{F}_{z,a_j} > \lambda_P$ yet not $\exists k\ \exists j\ E_z \cdot I_{z,a_k} \cdot \mathrm{F}_{z,a_j} \gg \lambda_P$. It is difficult to decide what constitutes $\gg$ versus $>$. If enough executions were used for both $\lambda_P$ and the estimates, then any difference between $E_z \cdot I_{z,a_k} \cdot \mathrm{F}_{z,a_j}$ and $\lambda_P$ greater than 0.05 should qualify as $\gg$; any difference less than 0.05 and greater than 0.02 may be considered $>$; any difference less than 0.02 may be considered as $\approx$.[16]

The equations given in this debugging algorithm assume independence between the infection estimate and the failure propagation estimate. The identical

[16] The values 0.05 and 0.02 are arbitrary.

situation arose in the discussion of determining the latent failure rate. In general, this assumption appears to be valid.[17] However it can be shown that there are cases for which this assumption does not hold. This assumption can allow for the mis-classification of locations; it may cause locations to be ignored during debugging due to the product of the infection, propagation, and execution estimates being larger than $\lambda_P$. In the worst case, input points which readily infect at a location through any semantic alternatives do not readily propagate for many active variable perturbations; for this phenomenon, subtraction of the proportion of "nonpropagating" points occurs when the latent failure rates are determined in equations 5.6 - 5.8 and equation 5.10; this leaves the potential of producing a negative result. Since

$$\forall\, \mathrm{F}_{z,a_j}\ \forall\, I_{z,a_k} \quad (\mathrm{F}_{z,a_j} \cdot I_{z,a_k}) \geq (I_{z,a_k} - (1 - \mathrm{F}_{z,a_j})) \qquad 0.0 \leq \mathrm{F}_{z,a_j}, I_{z,a_k} \leq 1.0,$$

the second debugging algorithm accounts for the non-independence that is ignored by the first algorithm. In general, the second debugging algorithm creates many more suspects than the first algorithm, however this algorithm is less likely to ignore locations where a large amount of type I cancellation occurs in their successor locations.[7]

The independence phenomenon creates difficulty in applying propagation and infection analysis as a debugging tool. By using the first algorithm, potential mis-classification of locations as not being suspects may occur. However acknowledgement that the independence assumption is rarely invalid but not completely invalid increases the number of locations classified as suspects, potentially to the point of saying every location is a suspect, which defeats the algorithm's purpose.

Regardless, the second debugging algorithm becomes:

1. to determine if a location $z$ is a *suspect*, take each $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair, get the latent failure rate, then multiply this product by $E_z$, and compare this product to $\lambda_P$. If

$$\exists k\ \exists j\ E_z \cdot (I_{z,a_k} - (1 - \mathrm{F}_{z,a_j})) \gg \lambda_P, \tag{5.37}$$

then an incorrect active variable $a_j$ and semantic alternative $a_k$ at location $z$ is not further considered as potentially causing $\lambda_P$; ignore the pairs satisfying

[17] From the tables in Chapter 6.

[7] This modification will potentially increase the number of suspect locations when the propagation estimates are small; if the propagation estimates are large, then this modification should have a minor effect. If the independence assumption between the infection estimates and propagation estimates is deemed valid, then the first algorithm should be applied.

inequality 6; they are not suspects,

2. If any $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair such that $\exists k\ \exists j\ E_z \cdot (I_{z,a_k} - (1 - \mathrm{F}_{z,a_j})) \approx \lambda_P$ is considered suspicious, thus location $z$ is a suspect. If every $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair at location $z$ is not suspicious, then location $z$ is not a suspect.

3. those $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pairs where $\exists k\ \exists j\ E_z \cdot (I_{z,a_k} - (1 - \mathrm{F}_{z,a_j})) \ll \lambda_P$ can not be ignored; by the single fault assumption, there may be several $(I_{z,a_k} - (1 - \mathrm{F}_{z,a_j}))$ summing up to $\approx \lambda_P$. The set of combinations where

$$\exists k_1\ \exists j_1\ E_l \cdot (I_{l,a_{k_1}} - (1 - \mathrm{F}_{l,a_{j_1}})) + \ldots + \exists k_n\ \exists j_n\ E_p \cdot (I_{p,a_{k_n}} - (1 - \mathrm{F}_{p,a_{j_n}})) \approx \lambda_P \tag{5.38}$$

is true therefore forces locations $l, \ldots, p$ to be considered as suspects.

4. the set of locations having at least one $\mathrm{F}_{z,a_j}$, $I_{z,a_k}$ pair considered as suspicious is the set of suspects; this is the set from which either static or dynamic analysis of the code can begin.

Enormous efficiency may be gained in both algorithms by keeping statistics on the most recently executed path or set of paths. This information immediately reduces the number of locations whose estimates are used in the previous rules.

The limitations to the proposed debugging algorithms are:

1. distributed faults,

2. if the fault is caused by a fault class not represented by the set of semantic alternatives,

3. if the perturbation function does not adequately represent the impacts of the potential faults, and

4. the size of $\lambda_P$; although locations have been successfully debugged producing a failure rate on the order of $10^{-6}$, it may be that the best that is achievable with propagation and infection analysis is on the order of $10^{-7}$; the lower bound limitations on $\lambda_P$ using propagation and infection analysis are yet undetermined.

Limitations (2) and (3) can be minimized in strength by increasing the scope of $\mathcal{F}$ and the potential faults represented by the perturbation function presented to **A** at a location. To minimize (1), a re-definition of what constitutes of location is required; this is a complex problem which is currently under investigation. The fourth limitation is speculative.

### 5.3.6 Testing Complexity

Halstead has developed one of the best known studies of software complexity [18]. Halstead uses a set of measures attained after the code is generated. These include the number of distinct operators, number of distinct operands, total number of operator occurrences, and total number of operand occurrences [18]. With these measures, Halstead develops equations for overall program length, program volume, and a measure of software complexity. The McCabe complexity measure is another well-known complexity measure developed from the control flowgraph of the program. McCabe defines a complexity measure based upon the cyclical complexity of a graph [18].

In general, the more complex software, the harder it will be to test. Propagation and infection analysis can be used for a testing complexity metric, $\Upsilon$, which describes the complexity involved in testing a program by producing the number of executions required to achieve a certain confidence in the assertion that the program is correct. $\Upsilon$ is a function of $\Delta$ and $PEC$. $\Upsilon$ is:

$$\Upsilon = \sum_{z \in \{PEC\}} \left\lceil \frac{\ln(1-\alpha)}{\ln(1-lfr_z)} \right\rceil . \tag{5.39}$$

As an example using equation 5.39, consider a program $P$ with four path equivalence classes $i$, $j$, $k$, and $l$, where $lfr_i = 0.01$, $lfr_j = 0.02$, $lfr_k = 0.009$, $lfr_l = 0.5$, and $\alpha = 0.999$. Then $\Upsilon \approx 1.8\text{x}10^3$. Now consider a program $K$ with one path equivalence class, and a latent failure rate of 0.00001. Then $\Upsilon \approx 6.9\text{x}10^5$. Although $P$ has more equivalence classes, it requires fewer test points since it exhibits a lesser tendency to hide faults. Program $K$, however, easily hides faults, therefore is harder for testers, and thus has a higher $\Upsilon$.

The metric in equation 5.39 refutes the notion that more locations mean higher testing complexity. This metric instead considers complexity to be relative to the minimum latent failure rate over the set of path equivalence classes. For non-positive latent failure rates, the corresponding path equivalence classes are ignored, and equation 5.39 must ignore path equivalence classes with these latent failure rates. This causes an "incomplete" complexity metric; investigation is continuing into other complexity metric models for path equivalence classes having non-positive latent failure rates that can be incorporated in equation 5.39. Once clear solution is to assume independence between the infection estimate and the propagation estimate, assuring a zero or greater latent failure rate. Experiments using propagation and infection analysis for debugging[12] have shown that in general the independence assumption between propagation and infection estimates holds. In fact, it may be extremely rare for the non-independence phenomenon to occur.

## 5.4 Conclusions

A new program characterization termed a software faultprint is introduced in Chapter 5. The algorithms for auxiliary processes for building the software faultprint are detailed. Chapter 5 has introduced application areas where software faultprints may be useful. This chapter has shown potential schemes for using the faultprint contents towards these applications. A summary of additional study needed on these areas follows:

**Probable Correctness** This model attempts to predict from a sequence of successful executions whether the next execution will be successful. This model's value is directly related to the measurement scheme of the minimum failure probability. For the minimum probability, the latent failure rate is used which is dependent on $\Delta$. Potentially, strict black-box testing is a better method for this estimate,

**Software Reliability** is the probability of failure-free operation of a computer program for a specified time in a specified environment. This thesis software reliability model, as well as all software reliability models, suffers from dependence on the input distribution. Time is ignored in this model, and reliability is per execution rather than per unit time. If the input distribution changes drastically, then the propagation and infection estimates are less meaningful, and may cause this model to be useless unless propagation and infection analysis is re-performed.

**Ultra-reliability** is the measurement of extremely high software reliability. This application is only mentioned because of propagation and infection analysis's ability to find locations that can have tiny impacts on the failure rate. These locations are a software tester's nightmare if ultra-reliable software is desired.

**Software Testing** This application is intuitive; empirical evidence is needed to determine the relationship between interfailure times for specific faults and the latent failure rates of the locations containing those fault.

**Debugging** is the application of determining where a particular fault is when failure is observed. This model is demonstrated in Chapter 6. The question persists as to whether infection and failure propagation estimates can be considered as independent.

**Testing Complexity Metric** is an application for determining how hard a particular program will be to test. This model suggests that the complexity of code is a function of how easily it is to hide faults, instead of some empirical characteristic such as the number of lines or modules. It is intuitive, however, that there may be a relationship between length of the program and $\Upsilon$.

Admittedly, the emphasis of the thesis has been in producing the software faultprint, not in applications of it. Hence the models presented are elementary and have not been substantiated through practice; they are included as potential foundations for additional work.

There are potentially other application areas not mentioned. An example is computer security. If a program has the ability to access information such as its value distributions during execution, then it could "self-test" the data states it was generating. If it detected a data state or series of data states not previously encountered, it could produce warnings about its output. This could be useful in the case of current operating system viruses. [19] gives evidence that for certain types of programs, the number of different data states produce in the operating environment is actually small. If true, the information could be used in this manner. To achieve such, research is needed into Method II value distribution creation; a realistic scheme for storing internal data states could make this a viable model for fighting "computer viruses".

# References

[1] C. V. Ramamoorthy and F. B. Bastani. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering SE-8*, pp. 543–371, July 1982.

[2] Lee J. White and Edward I. Cohen. A domain strategy fo computer program testing. *IEEE Transactions on Software Engineering*, SE-6:pp. 247–257, May 1980.

[3] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pp. 156–173, June 1975.

[4] Richard G. Hamlet. Probable correctness theory. *Information Processing Letters*, pp. 17–25, April 1987.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pp. 576–583, October 1969.

[6] Willam E. Howden. Completeness criteria for testing elementary program functions. *Proceedings Fifth International Conference on Software Engineering*, pp. 235–243, 1981.

[7] William E. Howden. Applicability of software validation techniques to scientific programs. *ACM Transactions on Programming Languages and Systems*, pp.307–320, July 1980.

[8] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, pp. 208–215, September 1976.

[9] J. C. Huang. An approach to testing. *ACM Computing Surveys*, pp. 113–128, September 1975.

[10] B. Littlewood. Theories of software reliability: how good are they and how can they be improved? *IEEE Transactions on Software Engineering SE-6*, pp. 489–500, September 1980.

[11] HARLAN D. MILLS. *Software Productivity.* Boston: Little, Brown, 1983.

[12] JEFFREY M. VOAS AND LARRY J. MORELL. Propagation and infection analysis(PIA) applied to debugging. *IEEE Southeastcon '90 Record*, April 1990.

[13] LARRY J. MORELL. *Unit Testing and Analysis SEI Curriculum Module SEI-CM-9-1.0.* Technical Report, Software Engineering Institute, Carnegie Mellon University, October 1987.

[14] LARRY JOE MORELL. *A Theory of Error-based Testing.* Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[15] MUSA AND IANNINO AND OKUMOTO. *Software Reliability Measurement Prediction Application.* McGraw-Hill, 1987.

[16] ELAINE J. WEYUKER AND THOMAS J. OSTRAND. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, pp. 236–246, May 1980.

[17] STEVE PARK. *Lecture notes on simulation, Unpublished.* College of William and Mary, August 1988.

[18] ROGER S. PRESSMAN. *Software Engineering A Practitioner's Approach.* McGraw-Hill Book Company, 1982.

[19] PETER W. PROTZEL. Automatically generated acceptance test: a software reliability experiment. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 196–203, July 1988.

[20] JOHN RUSHBY. *Quality Measures and Assurance for AI Software.* Technical Report NASA Contractor Report 4187, SRI International, Meno Park, CA., 1988.

[21] RICHARD A. DEMILLO AND RICHARD J. LIPTON AND FREDERICK G. SAYWARD. Hints on test data selection: help for the practicing programmer. *Computer*, pp. 34–41, April 1978.

[22] DICK HAMLET AND ROSS TAYLOR. Partition testing does not inspire confidence. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 206–215, July 1988.

[23] L.G. VALIANT. A theory of the learnable. *Communications of the ACM*, 27(11):pp.1134–1142, November 1984.

[24] K. MILLER AND L. MORELL AND R. NOONAN AND S. PARK AND B. MURRILL AND J. VOAS. Estimating the Failure Probability of Software When Testing Reveals No Errors. *IEEE Transactions on Software Engineering*, to be submitted 1990.

[25] LARRY J. MORELL AND JEFFREY M. VOAS. *Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability.* Technical Report WM-88-2, College of William and Mary, Department of Computer Science, September 1988.

# Chapter 6

# Empirical Results

Chapter 6 shows that the particular implementation of the propagation and infection analysis model defined in Chapters 2 and 3 produces the results for which the model was designed. The experimental results of Chapter 6 have been produced from "real-world" software with both artificially implanted faults and actual faults. The results are produced without an oracle, however there must exist a failure rate.

Chapter 6 is organized as follows: Section 6.1 informally shows that the class of faults, $\mathcal{F}$, described in Chapter 4 has enough breadth to include a small portion of the well-published set of faults in the "Common Blunders" chapter of [10]. Section 6.1 shows that the decision of the class of faults in $\mathcal{F}$ was not entirely arbitrary; there are common faults which it will catch. Section 6.2 shows that propagation and infection analysis has enough resolution in its point estimators to accurately quantify the impacts that a location has on the failure rate in order to use propagation and infection analysis as a debugging tool. Section 6.2 details several "blind" experiments of propagation and infection analysis being successfully applied to debugging.[1] Section 6.3 shows the similarity between the failure propagation estimates and infection estimates of successive versions of a program. This provides clues as to when propagation and infection analysis may begin during software development without the penalty of needing to reperform propagation and infection analysis.

[1] Debugging was chosen for substantiating the thesis since debugging is the easiest means for showing that an incorrect program's behavior with a fault was mirrored by propagation and infection estimates.

| *number* | *page* | *fault description* | *correction* |
|---|---|---|---|
| 1 | 77 | failure to initialize variable | |
| 2 | 78 | $(term < e)$ | $(abs(term) < e)$ |
| 3 | 78 | $-(1)^x$ | $(-1)^x$ |
| 4 | 79 | failure to initialize variable | |
| 5 | 79 | type problem | |
| 6 | 80 | label at wrong location | |
| 7 | 80 | mum= 0 | mum(i)= 0 |
| 8 | 81 | incorrect use of Fortran data construct | |
| 9 | 82 | if ctr > 45 | if ctr > 46 |
| 10 | 82 | ctr = 0 | ctr = 1 |
| 11 | 84 | $>$ | $\geq$ |
| 12<br>13 | 85 | if b+c<m goto L20 | if b+c<0.005 goto L10<br>else if b+c<m goto L20 |
| 14 | 87 | xlow, xhigh | x(low), x(high) |
| 15 | 87 | infinite loop | |
| 16 | 90 | marks(i) -1/10 + 1 | (marks(i)-1)/10 + 1 |
| 17 | 90 | $sumsq - (sumx^2/an))/an - 1.0$ | $(sumsq - (sumx^2/an))/(an - 1.0)$ |
| 18 | 91-93 | floating point hardware problem | |

Table 6.1: Faults of "Common Blunders"

## 6.1 $\mathcal{F}$ Versus "Common Blunders"

This section looks at the faults contained in the "Common Blunders" chapter in [10]; the faults contained are listed in Table 6.1 according to the page number, the fault, and in some instances the correct code. Section 6.1 gives an idea of the fault types that infection analysis simulates and the fault types which it does not. There are several faults in the "Common Blunders" chapter that are out of the scope of the thesis. For those faults which are in the "Common Blunders" chapter which are not handled by infection analysis, alternate techniques for detection of their presence are mentioned.

Seven of the eighteen published faults are not applicable to propagation and infection analysis: faults 8 and 17 are language problems, 13, 7, and 5 are compilation faults, fault 14 is an infinite loop, and fault 6 is a label in the wrong location. Faults 1 and 4 can be caught using dataflow analysis [5, 6, 4, 2] since this is a reference before definition problem. Faults 2 and 3 are not defined in $\mathcal{F}$; they require a more complex class of faults for both expression and predicate infection analysis. One problem with infection analysis, specifically $\mathcal{F}$, is faults such as 16 and 17 which are precedence faults; precedence faults are not included in $\mathcal{F}$. Process Simplify produces simple expressions with the incorrect precedence. Methods for handling precedent faults include requiring that each declared variable have a unit associated with it. An example is the expression $a * b * (c + d)$.

If a has units $m/s^2$, b has units $s$, c has units $s$, and d has units $m$, then this fault can not occur since c and d do not have the same units.[2]

Faults 9 and 10 together form a distributed fault. $\mathcal{F}$ is limited to single location faults; distributed faults are not considered during infection analysis due to the combinatorial problem of how to combine single locations into a distributed location. If failure propagation analysis perturbed combinations of active variables instead of single active variables in a single data state, distributed faults' impacts could be partially simulated. For example, in a four single location sequence, there are ten location combinations for creating a distributed fault: (1,2), (2,3), (3,4), (1,3), (1,4), (2,4), (1,2,3), (1,2,4), (2,3,4), (1,2,3,4).

Fault 11 is included in $\mathcal{F}$. More fault classes can be included in the infection analysis process by redefining $\mathcal{F}$. An explosion into the number of semantic alternatives results. One such modification to $\mathcal{F}$ might be to include compound faults such as fault 3 (this is two faults: $0 + (-1^x)$) into the definition of $\mathcal{F}$. Infection analysis estimates are a function of the semantic alternatives from $\mathcal{F}$, so it is important to include as many "potential" faults as is computationally feasible. If the semantic alternative class is small, the question might arise as to whether the class of faults is too restrictive, i.e., is there a class of frequently occurring faults with low infection estimates not represented in $\mathcal{F}$? In this example, several compound faults are found in the code segments from "Common Blunders " which are not represented in $\mathcal{F}$. $\mathcal{F}$ remains a parameter in infection analysis implementation and by increasing the breadth of $\mathcal{F}$, this infection analysis implementation becomes more meaningful.

## 6.2 Experiments for Validating the PIA Model and Implementation

This section contains experimental results that indicate that the propagation and infection analysis model and implementation are a plausible method for isolating locations that contain faults of specific impacts on the program output behavior. Each of the three programs used had a failure rate of less than $10^{-4}$ (requirement 2(d) of the definition of closely correct). Better yet, these propagation and infection analysis estimates closely mimic the observed behavior of the code. Three experiments were performed. Most of the experiments of this thesis were executed

[2] This problem is partially solved by requiring that variables have associated units; the Integrated Verification and Testing System (IVTS)[1] is an environment requiring units for programs in the language is Hal/S[3, 8].

on a SUN 3/50.[3]

Two participants were used in Experiments I and II, one participant to set up the experiment and one participant to perform it. Throughout Chapter 6, the person who set up the experiment is referred to as participant A, the other person is referred to as participant B. Note that these were "blind" experiments, meaning that participant A who inserted a fault never revealed to participant B where the fault was; only the locations for participant B to perform the analysis at. It was participant B's responsibility to apply propagation and infection analysis and attempt to determine where the fault occurred from the propagation and infection analysis estimates and the supplied failure rate.

### 6.2.1 Experiment I

The first program is found in Appendix C and is taken from [12]. A fault was placed into the sqr call in the statement

```
t:= 0.9*(1.0+sqr(y))*exp(em*glalxm-gammln(em+1.0)-glg);
```

producing

```
t:= 0.9*(1.0+sqr(1.0+y))*exp(em*glalxm-gammln(em+1.0)-glg);
```

by participant A. The failure rate observed by participant A from this fault is $2.0 * 10^{-6}$. Participant B performed failure propagation analysis on this non-simplified expression and received a high failure propagation estimate. Hence participant A could not locate the fault from the failure propagation analysis on the complex expression. The complex expression was then broken up by participant A for participant B into the following simple expressions:

```
aa := sqr(y+1.0); {1}
bb := aa + 1.0; {2}
cc := em + 1.0; {3}
dd := gammln(cc); {4}
ee := em*glalxm-dd-glg; {5}
ff := bb * exp(ee); {6}
```

[3] SUN 3/50 is a trademark of Sun Microsystems.

| *location* | *propagation estimate* |
|---|---|
| 1 | 7.664e-02 |
| 2 | 3.9831e-01 |
| 3 | 8.5781e-01 |
| 4 | 8.525e-01 |
| 5 | 1.3581e-01 |
| 6 | 3.9687e-01 |

Table 6.2: Failure propagation estimates for *poidev* using uniform(12, 13) for inputs

| *location* | *propagation estimate* |
|---|---|
| 1 | 3.0e-05 |
| 2 | 3.8917e-01 |
| 3 | 8.7326e-01 |
| 4 | 8.7613e-01 |
| 5 | 6.176e-02 |
| 6 | 3.9027e-01 |

Table 6.3: Failure propagation estimates for *poidev* using uniform(12, $10^6$) for inputs

```
t := 0.9 * ff; {7}
```

Participant B is given the observed failure rate, the entire program with the simple expressions, the input distribution producing the $2.0 * 10^{-6}$ failure rate, and a list of locations where the fault was potentially residing. In the experiment, participant B is told the fault is in one of the first six simple locations created from the complex expression, and that the input distribution is uniform(13.0,1000000.0).

The failure propagation estimates participant B produced when he limited the input distribution to uniform(12.0, 13.0) are found in Table 6.2. These estimates were obtained from only perturbing the active variable receiving the action at that location; this reduction in the amount of perturbation analysis also applies to the estimates found in Table 6.3 which participant B produced from an input distribution of uniform(12.0, 1000000.0). From these failure propagation estimates, participant A postulates that location 1 contains the fault, which indeed it did. Hence participant B was able to accurately and positively identify the subexpression in the expression causing the observed failure rate. No infection analysis is performed and the experiment was halted; this experiment had shown two abilities of failure propagation analysis:

| *location* | *propagation estimate* |
|---|---|
| 1 | 0.2818 |
| 2 | 0.0 |
| 3 | 1.0 |
| 4 | 0.227199512178 |
| 5 | 0.273673449675 |

Table 6.4: Failure propagation estimates for *qcksrt* using perturb(0.95, 1.05,x)

1. simpler expressions do give more precise failure propagation estimates, and
2. these failure propagation estimates do mimic the effects of faults.

To be complete, for a full debugging experiment, infection analysis should be performed, and the conclusions about suspects determined according to the algorithms provided in Chapter 5. This experiment was performed in the early stages of building the thesis model to validate the notion of failure propagation analysis.

### 6.2.2 Experiment II

The second experiment is also taken from [12], specifically *procedure qcksrt* on pages 726-727. Five locations were identified by participant A to participant B as potentially containing the fault. The locations are notated and the program used in performing the failure propagation analysis by participant B is included in Appendix B. The results participant B found from failure propagation analysis after 250,000 input points from the input distribution that produced a $7.5 * 10^{-5}$ failure rate are found in Table 6.4. Again as in Experiment I, only the active variable receiving the action at the location is perturbed; the same is true for the estimates in Table 6.5. The perturbation function used for the estimates of Table 6.4 was uniform(0.95x, 1.05x). From these failure propagation estimates, nothing specific can be stated except that a pseudo-failure from perturbing at location 2 never occurred in 250,000 trials.[4] Table 6.5 contains the failure propagation estimates with a different perturbation function, namely uniform(0.5x, 1.5x).

Table 6.8 gives the semantic alternatives participant B used during infection analysis at the five locations with the corresponding infection estimates. Note that

[4] The explanation why location 2 has no failure propagation is that it is a pseudo-random number generator that quicksort uses. Hence an infection only decreases randomness, not correctness.

| *location* | *propagation estimate* |
|---|---|
| 1 | 0.369008 |
| 2 | 0.0 |
| 3 | 1.0 |
| 4 | 0.346619520890 |
| 5 | 0.188418873305 |

Table 6.5: Failure propagation estimates for *qcksrt* using perturb(0.5, 1.5,x)

this program is not in simplified expression form, and not all possible elements of $\mathcal{F}$ were used. This reduced the computing needed.[5]

For participant B, there are three noticeable estimates in Table 6.8: 0.0000101, 0.0001462, 0.000009746 as notated by asterisks. The third estimate (0.000009746) occurs twice for two different semantic alternatives. The goal for participant B is to find an infection estimate for a location that when multiplied by the corresponding failure propagation estimate of the location approximates the failure rate that participant A supplied (the execution estimate at a location should also be considered if it is less than 1.0, however in this case each of these five locations is always executed). For this case, $0.0001462 * 0.3466 = 0.0000506 \approx 0.000075$. And indeed location 4 is where the fault was inserted, hence participant B again succeeded. Not only is location 4 the location of the fault, but the semantic alternative causing the infection estimate of 0.0001462 is the fault. This semantic alternative was found by taking the value distribution for variable l and picking the value with the highest probability of occurring, which in this case was the constant 1. So substituting variable l for the constant 1 was the fault participant A inserted, and the low failure rate occurred because the probability density function of l preceding the location under analysis was drastically spiked at the value of 1.

### 6.2.3 Experiment III

The final experiment is performed on a version from the $n$-version LIC experiment discussed in [11]. The version is written in Fortran-77 and was approximately 500 lines in length. The particular version used had an seemingly zero failure rate after

[5] A SUN 3/50 is truly incapable of performing complete propagation and infection analysis except for trivial input programs. This was the main computing resource available to the author.

a huge number of executions to determine the failure rate.[6] Experiment III differs from the previous two experiments because the fault causing the observed failure rate was the actual fault in the code, not a fault inserted by either participant A or participant B. Hence this is a real program with a real fault, not a real program with a "inserted" fault.

After $2 \cdot 10^6$ executions of the LIC version on a Convex-220, the failure rate was determined to be 0.0, since no failures occurred. Hence this program also satisfies criteria 2(d) of the closely correct criterion. Eight randomly selected locations were noted for failure propagation and infection analysis, including the location known to contain the fault.[7] The incorrect code of this location follows:

```
IF ((norm1 .EQ. 0.0) .OR. (norm2 .EQ. 0.0)) THEN
  angle5 = 0.0
ELSE
  angle5 = ACOS((xone*xthree + yone*ythree)/(norm1*norm2))
END IF
```

The correct code at the location should be:

```
IF ((norm1 .EQ. 0.0) .OR. (norm2 .EQ. 0.0)) THEN
  angle5 = 0.0
ELSE
  csn = (xone*xthree + yone*ythree)/(norm1*norm2)
  IF (csn .LT. -1.0) THEN
    csn = -1.0
  ELSE IF (csn .GT. 1.0) THEN
    csn = 1.0
  END IF
  angle5 = ACOS(csn)
END IF
```

Table 6.6 contains the failure propagation estimates for the active variable being assigned at the eight locations. For the location actually containing the fault, a

[6] There existed an oracle for the version being analyzed which was needed to determine the version's failure rate. The fault location was pre-known as well as the fact that this was the only fault in the program.

[7] The fault is a missing path, so more than one location is missing. Since the net effect of missing code is a changed assigned value, propagation analysis on the active variable being defined should catch this missing path.

| *location* | *propagation estimate* |
|---|---|
| 1 | 0.25023 |
| 2 | 0.0 |
| 3 | 0.0 |
| 4 | 1.0 |
| 5 | 0.0416304 |
| 6 | 0.0416304 |
| 7 | 0.0 |
| 8 | 0.0 |

Table 6.6: Failure propagation estimates for LIC version

| *location* | *original code* |
|---|---|
| 1 | e=25. |
| 2 | i=i+1 |
| 3 | temp+parray(ii,j) |
| 4 | temp+fum(i) |
| 5 | ((x2-px)**2 + (y2-py)**2)) |
| 6 | ((slope*x1 + x2/slope - y1 +y2)*slope)/(slope**2 +1)) |
| 7 | (x1-x2)**2 +(y1-y2)**2) |
| 8 | ACOS((xone*xthree + yone*ythree)/(norm1*norm2)) |

Table 6.7: Original code from which the semantic alternatives were derived for the LIC version

perturbation function is applied on the current value of the variable being defined which is not properly being defined due to the missing path.[8] Again, perturbing on each active variable at the location is not performed to limit the computational costs and manual costs. The infection estimates for a subset of the recommended semantic alternative set[9] with the alternatives are found in Table 6.9.

Notice in Table 6.9 that there are no infection estimates on the order of $\ll 2 * 10^{-6}$.[10] In this version, $E_l$ is high for each of the eight locations, i.e., approximately 1.0. Hence the failure propagation estimates are all that is left in trying to determine which locations are suspects in this example.

[8] The fault actually is just range checking on the value of (xone*xthree + yone*ythree)/(norml*norm2) which is virtually never required.

[9] According to the definition of $\mathcal{F}_l$ in Chapter 4.

[10] Note that for the infection analysis estimates shown in Table 6.9, where $10^6$ input points were used, the cost of this analysis was 267.4 cpu seconds on a Convex 220; had this analysis been done for all 500 locations, it would have taken approximately 4 hours (assuming complete infection analysis using the sequential pseudo-code algorithm on the Convex 220 after the commented locations are subtracted).

From Table 6.6, there are four locations which show no failure propagation after a uniform perturbing distribution and perturbation function parameters of (-2.0, 2.0) are applied after executing $10^4$ inputs;[11] these are locations 2, 3, 7, and 8. It is unusually strange to find 4 out of eight locations displaying no failure propagation after $10^4$ perturbations at each location. As it turns out, the specification for this LIC version is one which contains many computations and only a few single bits of output. This specification allows for versions with a large amount of type I cancellation.

The location with the fault causing the zero failure rate was location 8, which was one of the four locations classified as suspects due to a zero failure propagation estimate at location 8. So debugging was "somewhat" successful in identifying the location with the fault as being a suspect; propagation analysis was very successful in showing that its estimates do reflect the output behavior of each location of a program.

Experiment III has shown two remarkable results:

1. It has isolated a location which contained a fault, even though the program had never failed after $2 \cdot 10^6$ inputs, and
2. Experiment III showed that failure propagation analysis does have the ability to detect missing code at a location by perturbing on each active variable.

From (1), it has been shown how propagation and infection analysis can be applied to ultra-reliable software, particularly software which has never failed. The second result shows that propagation and infection analysis has a unique ability to detect missing paths which no other fault detection scheme has.

[11] Several different perturbation functions were applied to the LIC version in an attempt to force more pseudo-failures. In this experiment, (-2.0,2.0) perturbation function parameters with the uniform perturbing distribution caused frequent run-time termination due to the fact that the version contains a lot of sqrt calls and trigonometric function calls. Such occurrences can help dictate the infection interval if the code is statically analyzed for such function calls.

| *semantic alternative* | *location* | *infection estimate* |
|---|---|---|
| ((ir-l) > m) | 1 | 7.23907390917186e-01 |
| ((l-ir) < m) | 1 | 3.21905609973286e-01 |
| ((ir-l) = m) | 1 | 9.54186999109528e-01 |
| ((ir-l) <= m) | 1 | 2.76092609082814e-01 |
| ((ir-l) >= m) | 1 | 1.0 |
| ((ir-l) < fm) | 1 | 3.21905609973286e-01 |
| ((ir-l) < fmi) | 1 | 5.20975957257346e-01 |
| ((ir) < m) | 1 | 3.91337488869101e-01 |
| ((l) < m) | 1 | 4.66368655387355e-01 |
| ((-l) < m) | 1 | 3.21905609973286e-01 |
| ((ir+l) < m) | 1 | 4.47066785396260e-01 |
| ((ir*l) < m) | 1 | 3.91376669634907e-01 |
| ((ir-1) < m) | 1 | 3.21866429207480e-01 |
| ((jstack-l) < m) | 1 | 3.21905609973286e-01 |
| ((istack[jstack]-l) < m) | 1 | 4.99348174532502e-01 |
| ((arr[i]-l) < m) | 1 | 3.06924309884239e-01 |
| ((arr[i+1]-l) < m) | 1 | 3.57292965271594e-01 |
| ((ir-jstack) < m) | 1 | 2.77503116651825e-01 |
| ((ir-fmi) < m) | 1 | 3.21866429207480e-01 |
| ((ir-arr[i]) < m) | 1 | 3.61125556544969e-01 |
| ((ir-arr[i+1]) < m) | 1 | 3.87928762243989e-01 |
| ((ir-istack[jstack]) < m) | 1 | 2.46874443455031e-01 |
| ((ir-l) < jstack) | 1 | 4.17809439002671e-01 |
| ((ir-l) < arr[i]) | 1 | 3.05556544968833e-01 |
| ((ir-l) < arr[i+1]) | 1 | 2.54714158504007e-01 |
| ((ir-l) < istack[jstack]) | 1 | 3.56188780053428e-01 |
| ((1-l) < m) | 1 | 3.21905609973286e-01 |
| ((ir-a) < m) | 1 | 3.76463045414069e-01 |
| ((ir-l) < a) | 1 | 3.04341941228851e-01 |
| ((ir-l) < n) | 1 | 3.21905609973286e-01 |
| ((ir-n) < m) | 1 | 3.21905609973286e-01 |
| ((n-l) < m) | 1 | 3.16552092609083e-01 |
| (fx*fa-fc)/fm | 2 | 1.0 |
| (fa*fa-fc)/fm | 2 | 1.0 |
| (fa-fa-fc)/fm | 2 | 1.0 |
| (5*fa-fc)/fm | 2 | 1.0 |
| (fa-fc)/fm | 2 | 1.0 |
| (fm*fa-fc)/fm | 2 | 1.0 |
| (fx/fa-fc)/fm | 2 | 1.0 |
| (fx*fa*fc)/fm | 2 | 1.0 |
| (fx*fa*fc)+fm | 2 | 1.0 |
| (fx*fa*fc)*fm | 2 | 1.0 |
| fm/(fx*fa+fc) | 2 | 1.0 |
| (fx*fa+fc)*fx | 2 | 1.0 |
| (fx*fa+fc)*fa | 2 | 1.0 |
| (fx*fa+fc)*fc | 2 | 1.0 |
| (i*fa+fc)/fm | 2 | 1.0 |
| (fx*i+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+i)/fm | 2 | 1.0 |
| (fx*fa+fc)/i | 2 | 1.0 |

Table 6.8: Infection estimates for *qcksrt*

| *semantic alternative* | *location* | *infection estimate* |
|---|---|---|
| (l*fa+fc)/fm | 2 | 1.0 |
| (fx*l+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+l)/fm | 2 | 1.0 |
| (fx*fa+fc)/l | 2 | 1.0 |
| (j*fa+fc)/fm | 2 | 1.0 |
| (fx*j+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+j)/fm | 2 | 1.0 |
| (fx*fa+fc)/j | 2 | 1.0 |
| (ir*fa+fc)/fm | 2 | 1.0 |
| (fx*ir+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+ir)/fm | 2 | 1.0 |
| (fx*fa+fc)/ir | 2 | 1.0 |
| (fx*fx+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+fx)/fm | 2 | 1.0 |
| (fx*fa+fc)/fx | 2 | 1.0 |
| (fa*fa+fc)/fm | 2 | 1.0 |
| (fx*fa+fa)/fm | 2 | 1.0 |
| (fx*fa+fc)/fa | 2 | 1.0 |
| (fc*fa+fc)/fm | 2 | 1.0 |
| (fx*fc+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+fc)/fc | 2 | 1.0 |
| (fm*fa+fc)/fm | 2 | 1.0 |
| (fx*fm+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+fm)/fm | 2 | 1.0 |
| (iq*fa+fc)/fm | 2 | 1.0 |
| (fx*iq+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+iq)/fm | 2 | 1.0 |
| (fx*fa+fc)/iq | 2 | 1.0 |
| (fmi*fa+fc)/fm | 2 | 1.0 |
| (fx*fmi+fc)/fm | 2 | 1.10650069156293e-05* |
| (fx*fa+fmi)/fm | 2 | 1.0 |
| (fx*fa+fc)/fmi | 2 | 1.0 |
| (a*fa+fc)/fm | 2 | 9.02627939142462e-01 |
| (fx*a+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+a)/fm | 2 | 1.0 |
| (fx*fa+fc)/a | 2 | 1.0 |
| (arr[iq]*fa+fc)/fm | 2 | 8.76813278008299e-01 |
| (fx*arr[iq]+fc)/fm | 2 | 1.68077455048409e-02 |
| (fx*fa+arr[iq])/fm | 2 | 1.0 |
| (fx*fa+fc)/arr[iq] | 2 | 1.0 |
| 1 | 3 | 7.84467228610191e-01 |
| a-1 | 3 | 1.0 |
| i | 3 | 7.76794253052510e-01 |
| j | 3 | 7.34959855881503e-01 |
| 1-a | 3 | 1.0 |
| 5*a + 1 | 3 | 1.0 |
| arr[iq] | 3 | 4.85910637635389e-01 |
| iq | 3 | 7.84489469119053e-01 |
| arr[l] | 3 | 4.85910637635389e-01 |
| l | 3 | 7.84489469119053e-01 |

Table 6.8: Cont'd

| *semantic alternative* | *location* | *infection estimate* |
|---|---|---|
| arr[j] | 3 | 4.18010364077130e-01 |
| n | 3 | 9.77047794853546e-01 |
| fx | 3 | 1.0 |
| j-1 | 4 | 1.46203105353958e-04 |
| j+1 | 4 | 1.0 |
| j+l | 4 | 1.0 |
| l-j | 4 | 1.0 |
| j*l | 4 | 1.0 |
| j div l | 4 | 1.0 |
| l | 4 | 1.0 |
| 1 | 4 | 1.0 |
| j | 4 | 1.0 |
| j-j | 4 | 1.0 |
| 5*j - 3*l | 4 | 1.0 |
| j*j-2*l-3 | 4 | 1.0 |
| j-arr[i] | 4 | 9.74687369026385e-06* |
| j-i | 4 | 1.0 |
| j-ir | 4 | 1.0 |
| j-istack[jstack] | 4 | 1.0 |
| j-istack[jstack-1] | 4 | 1.0 |
| arr[i]-l | 4 | 7.51308517792918e-01 |
| i-l | 4 | 1.0 |
| ir-l | 4 | 5.95748413696307e-01 |
| istack[jstack]-l | 4 | 9.74687369026385e-06* |
| istack[jstack-1] -l | 4 | 8.73290642026570e-01 |
| l-l | 4 | 1.0 |
| i | 5 | 1.0 |
| i+1 | 5 | 1.0 |
| i+ir | 5 | 1.0 |
| ir-i | 5 | 7.41212838990706e-01 |
| i*ir | 5 | 1.0 |
| i div ir | 5 | 8.44691571150410e-01 |
| ir | 5 | 1.0 |
| 1 | 5 | 7.36053957484393e-01 |
| i | 5 | 1.0 |
| i-i | 5 | 8.44691571150410e-01 |
| i*i-i-1 | 5 | 7.36053957484393e-01 |
| i+i | 5 | 1.0 |
| i-jstack | 5 | 1.0 |
| i-arr[i] | 5 | 7.61226087480178e-01 |
| i-a | 5 | 7.61226087480178e-01 |
| i-n | 5 | 1.0 |
| i-ir | 5 | 1.0 |
| jstack-1 | 5 | 7.36053957484393e-01 |
| arr[i]-1 | 5 | 7.83186462452576e-01 |
| a-1 | 5 | 7.83186462452576e-01 |
| n-1 | 5 | 1.0 |
| ir-1 | 5 | 1.0 |

Table 6.8: Cont'd

| *location* | *semantic alternative* | *infection estimate* |
|---|---|---|
| 1 | 24 | 1.0 |
| 1 | L | 1.0 |
| 1 | Epsilon1 | 1.0 |
| 1 | 23 | 1.0 |
| 1 | Epsilon2 | 1.0 |
| 1 | 20 | 1.0 |
| 1 | l+R | 1.0 |
| 1 | N3 | 1.0 |
| 1 | M | 1.0 |
| 1 | 0 | 1.0 |
| 2 | (i-1) | 1.0 |
| 2 | 1 | 1.0 |
| 2 | 0 | 1.0 |
| 2 | 2 | 0.1164889 |
| 2 | l | 1.0 |
| 2 | i+n1+1 | 1.0 |
| 2 | i+n1-1 | 1.0 |
| 2 | p-n1-1 | 1.0 |
| 2 | y(i) | 1.0 |
| 2 | cmm(13) | 1.0 |
| 4 | (temp-fum(i)) | 1.0 |
| 4 | (fum(i)-temp) | 1.0 |
| 4 | temp | 1.0 |
| 4 | fum(i) | 1.0 |
| 4 | pum(i)+temp | 1.0 |
| 4 | temp-size | 1.0 |
| 4 | fum(i)+size | 0.9333333 |
| 4 | pum(i) | 1.0 |
| 4 | pum(i)-1 | 1.0 |
| 4 | pum(i)*temp | 1.0 |

Table 6.9: Infection estimates for LIC

| *location* | *semantic alternative* | *infection estimate* |
|---|---|---|
| 5 | (sqrt((x1-px)**2 + (y2-py)**2)) | 1.0 |
| 5 | (sqrt((y2-px)**2 + (y2-py)**2)) | 1.0 |
| 5 | (sqrt((x2-px)**2 + (x2-py)**2)) | 1.0 |
| 5 | (sqrt((x2-px)**2 + (y2-px)**2)) | 1.0 |
| 5 | (sqrt((x2-px)**2 + (y2-py))) | 1.0 |
| 5 | (sqrt((x1-px)**2 + (y2-py)**2)) | 1.0 |
| 5 | (sqrt((x2-px) + (y2-py)**2)) | 1.0 |
| 5 | (sqrt((x2-px) + (x2-py)**2)) | 1.0 |
| 5 | (sqrt((x2-px)**2 + (y2-py+1)**2)) | 1.0 |
| 5 | (sqrt((x2-px)**2 + (y2-py)**4)) | 1.0 |
| 6 | (((slope*y1 + x2/slope - y1 +y2)*slope)/(slope**2 +1)) | 0.9997583 |
| 6 | (((slope*x1 + x3/slope - y1 +y2)*slope)/(slope**2 +1)) | 1.0 |
| 6 | (((slope*x1 + x2/slope + y1 +y2)*slope)/(slope**2 +1)) | 1.0 |
| 6 | (((slope*x1 + x2/slope - y1 -y2)*slope)/(slope**2 +1)) | 1.0 |
| 6 | (((slope*x1 + x2/slope - y1 +y2)*slope)/(slope**2 )) | 1.0 |
| 6 | (((slope*y1 + x2/slope - y1 +y2)*slope)/(slope**2 )) | 1.0 |
| 6 | (((slope*x1 + x3/slope - y1 +y2)*slope)/(slope**2 )) | 1.0 |
| 6 | (((slope*x1 + x2/slope + y1 +y2)*slope)/(slope**2 )) | 1.0 |
| 6 | (((slope*x1 + x2/slope - y1 -y2)*slope)/(slope**2 )) | 1.0 |
| 6 | (((slope*x1 + x2/slope - y1 +y2)*slope)/(slope**2 )) | 1.0 |
| 7 | ((x1-x2)**2+(y1-y2)**3) | 1.0 |
| 7 | ((x1)**2+(y1-y2)**2) | 1.0 |
| 7 | ((x1-x2)**2+(y1+y2)**2) | 1.0 |
| 7 | ((y1-x2)**2+(y1-y2)**2) | 1.0 |
| 7 | ((x1-x2)**2+(x1-y2)**2) | 1.0 |
| 7 | ((x1-x2)**2+(y1-y2)**3) | 1.0 |
| 7 | ((x1-x2)**3+(y1-y2)**2) | 1.0 |
| 7 | ((x1-x2)**2+(y1-y1)**2) | 1.0 |
| 7 | ((x1-x2)**2-(y1-y2)**2) | 1.0 |
| 7 | ((x1-x2)**2+(y1*y2)**2) | 1.0 |
| 8 | (ACOS((xone*xthree * yone*ythree)/(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree + yone*ythree)/(norm2*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree + yone*ythree)*(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone + yone*ythree)/(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree - yone*ythree)/(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree / yone*ythree)/(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree + yone*ythree)/(norm2*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree + yone*ythree)*(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone + yone*ythree)/(norm1*norm2))) | 1.0 |
| 8 | (ACOS((xone*xthree - yone*ythree)/(norm1*norm2))) | 1.0 |

Table 6.9: Cont'd

## 6.3 Similarity of Propagation and Infection Estimates Between Successive Versions

Section 6.3's results are based upon the infection and failure propagation estimates gathered from a program referred to as Quadratic. The results presented are from failure propagation analysis and infection analysis. Realizing that viral propagation is a modified scheme of failure propagation through one location suggests that these results for failure propagation estimates will hold for viral propagation estimates.

The graphs shown throughout this chapter were created from both faults of large and small size. *Fault size* is a measure of the proportion of input points that reach the fault and cause failure. It is a function of the sampling scheme of the input distribution. For Quadratic, it was difficult to insert faults that affected few input points. Hence the graphs are derived from faults of relatively large size.

Consider Quadratic:

```
procedure Quadratic(a,b,c : real;
          var root1, root2 : real;
          var rootexists : boolean);
var
          d : real;
begin                                {location}
          d := b*b-4*a*c;            {1}
          if d < 0 then              {2}
             rootexists := false     {3}
          else
          begin
             rootexists := true;     {4}
             d := sqrt(d);           {5}
             root1 := -(b-d)/(2*a);  {6}
             root2 := -(b+d)/(2*a);{7}
          end;
end;
```

This program is correct for the following specification: If $a, b$, and $c$ are real number constants with $a \neq 0$ and $b^2 - 4ac \geq 0$, and if $x$ is a variable with domain being the set real numbers, then the solution set of $ax^2 + bx + c = 0$ is

$$\left\{\frac{-b+\sqrt{b^2-4ac}}{2a}, \frac{-b-\sqrt{b^2-4ac}}{2a}\right\}$$

[7]. Produce a program which assigns the solutions of the quadratic equation to variables root1 and root2 and set rootexists to true iff $b^2 - 4ac \geq 0$ regardless of whether $a \neq 0$; otherwise set rootexists to false.

The infection and failure propagation graphs throughout this chapter[12] have an abscissa-axis ranging between 0% and 100% representing infection and failure propagation values with probabilities between 0 and 1.0. The ordinate-axis represents the probability density of choosing an infection or failure propagation estimate of that probability. The appendix in [9] contains the program shell used to determine $P_x$'s infection and failure propagation curves. Each curve represents a summary of all propagation estimates over all locations in Quadratic. For the graphs where only one line appears, it is because all curves are sitting directly over one another.

Let $[P_1, P_2, P_3, ...]$ represent the sequence of unique versions of program $P$ over time, where $P_t$ is the successor version of $P_{t-1}$. The first three failure propagation graphs show the distribution of failure propagation estimates for versions $P_x, P_{x-1}, P_{x-2}(P_{x-1}$ has the remaining fault to be found, $P_{x-2}$ has the remaining 2 faults). In each graph in this chapter, the darkest curve represents version $P_x$, the thin dotted curve represents version $P_{x-2}$, and the medium thick solid curve represents version $P_{x-1}$ as illustrated by the insert in Figure 6.1. For the graphs of $P_{x-1}$ and $P_{x-2}$, there were three versions each, so that several fault combinations were used for the graphs. The faults were inserted manually. For the three $P_{x-1}$ versions, one fault was inserted into three copies of $P_x$, and for the three $P_{x-2}$ versions, another fault is inserted into 3 copies of $P_{x-1}$.

There is very little difference in the curves in the first three failure propagation graphs, Figure 6.1, Figure 6.2, and Figure 6.3. All value perturbations are performed with uniform(0.6*x, 1.4*x). Failure graph 1 shows the most diversity, with the very thin line representing no failure propagation. To explain this, consider the faults used: for $P_{x-1}$ the inserted fault was at location 7 and was -(b+a)/2*a; for $P_{x-2}$ there is an additional fault at location 5 which was d := 100. Since the fault at location 7 is the very last statement executed for the trip which encompasses it, failure propagation estimates for $P_x$ and $P_{x-1}$ are identical. However for $P_{x-2}$,

[12] The curves are slightly misleading so a note on them: the lines between any 2 locations does not imply a continuous line; the lines interpolate discrete points; each $x$ value is discrete and there is no continuity along the $x$-axis.

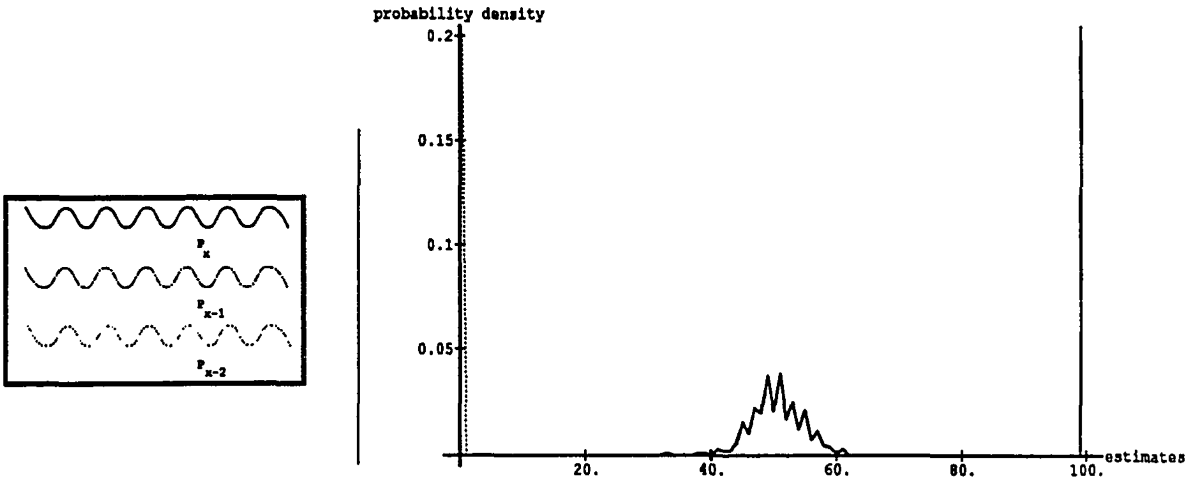


Fig. 6.1: Failure propagation estimates graph 1

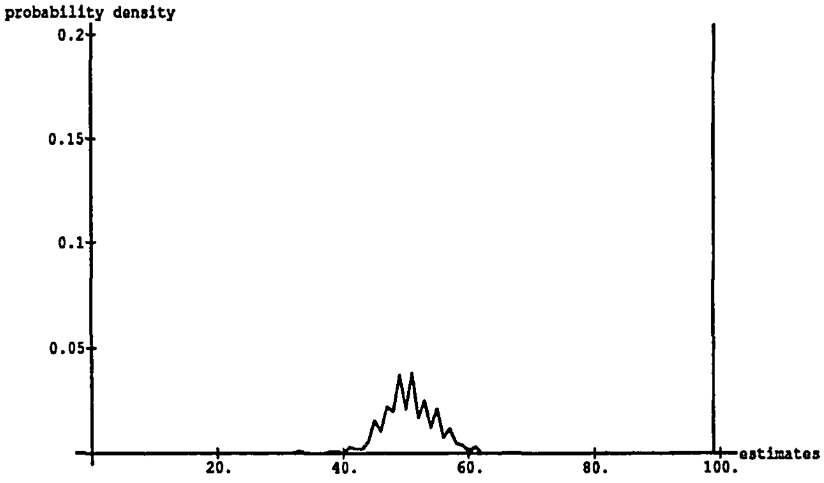


Fig. 6.2: Failure propagation estimates graph 2

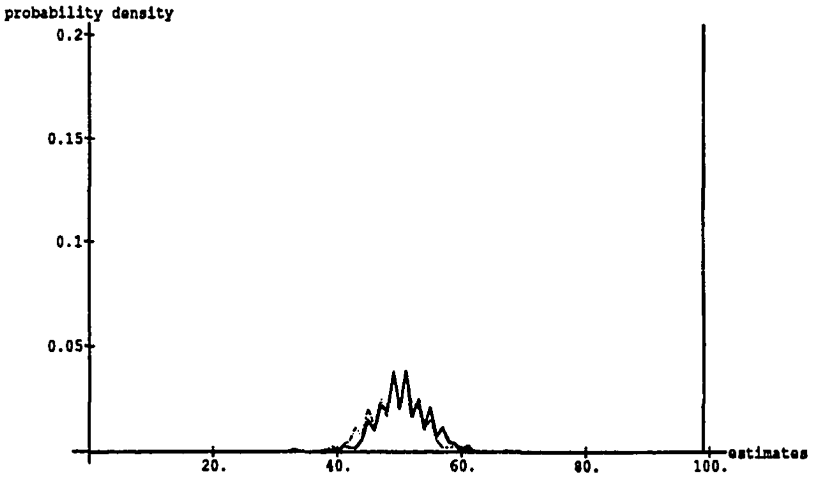


Fig. 6.3: Failure propagation estimates graph 3

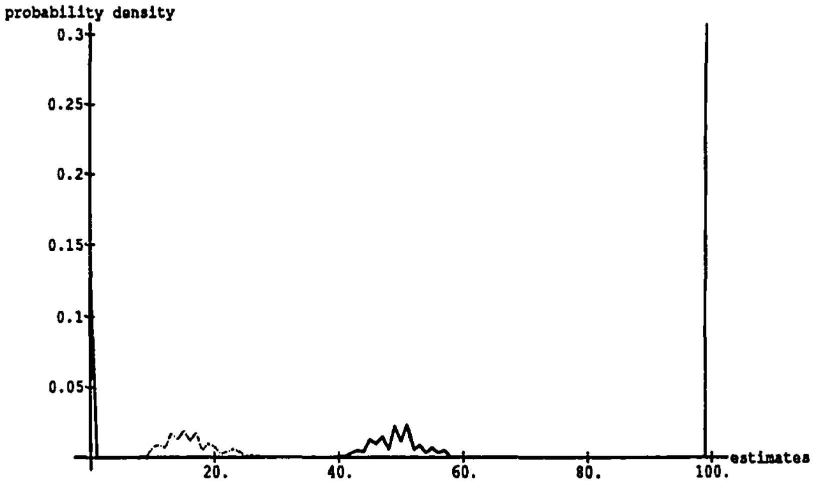


Fig. 6.4: Infection estimates graph 1

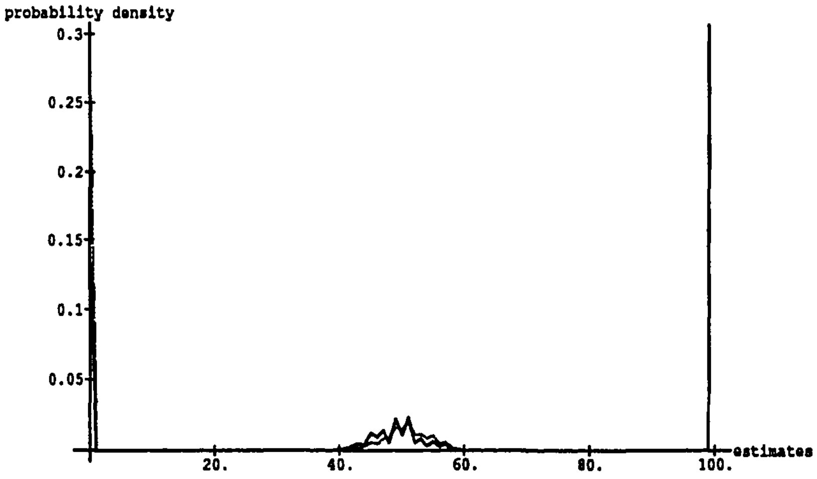


Fig. 6.5: Infection estimates graph 2

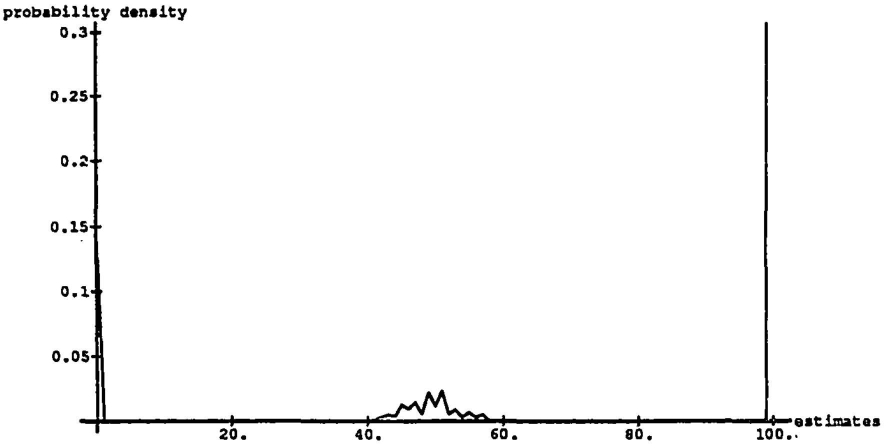


Fig. 6.6: Infection estimates graph 3

its curve represents many non-propagating points because of the fault at location 5; it succeeds any of the previous faults that are simulated by the perturbation function thus eliminating their effect. Any perturbed value of d at previous locations is nullified by this mutation making those viruses not propagate to failure. This leads to the following observation on the failure propagation algorithm.

Observation 6.1 *Let $z$ be the last location executed on a path, $x$ be a location where perturbation is being analyzed, and location $y$ be in the subpath from location $x$ to location $z$. There is no propagation from location $x$ to location $z$ if a virus inserted by a perturbation function at location $x$ is completely canceled(type I) by the action at location $y$. An example is when the action at location $y$ is a definition of the variable perturbed in the succeeding data space of location $x$, and the definition at location $y$ if not a function of the perturbed value at location $x$.*

$P_{x-1}$'s curve in failure propagation graph 2 was made by inserting if d <= 0 then at location 2. $P_{x-2}$'s curve in failure propagation graph 2 was made by inserting the additional fault b - 4*a*c at location 1. All three curves in this plot are identical. In failure propagation graph 3, $P_{x-1}$'s curve in failure propagation graph 2 was made by inserting d:=sqr(d) at location 5. $P_{x-2}$'s curve in failure propagation graph 2 was made by inserting the additional fault 4*a*c-b*b at location 1. $P_x$ and $P_{x-1}$ in this plot have identical curves, with $P_{x-2}$ practically identical. The versions represented in failure propagation graphs 1, 2, and 3 received the same input data.

The three versions represented in failure propagation graph 4 received the same input data, however it is a different input set than that used in the previous failure propagation graphs. It is important that the curves be similar for various input data as well which is noticeable by comparing the curves for $P_x$ in Figures 6.8 and 6.4. It is true that failure propagation estimates are a function in some sense of the input used, however it should not be the case that two sets of inputs produce drastic differences solely because the input sets are different. If this were true then failure propagation estimates would be extremely dependent upon what input data is used; this would be catastrophic for propagation and infection analysis. The results in failure propagation graph 4 are identical curves. The perturbation function used is uniform(0.3*x, 1.7*x), and the inserted faults are if d < 1 then at location 2 and root1 := -(d-b)/(2*a) at location 6. Failure propagation graph 5 also had the same input set as that for failure propagation graph 4 presented to its three versions. All three versions have the same curve. The perturbation function is also uniform(0.3*x, 1.7*x); the faults are rootexists := true at location 3 and d := sqr(d) at location 5.

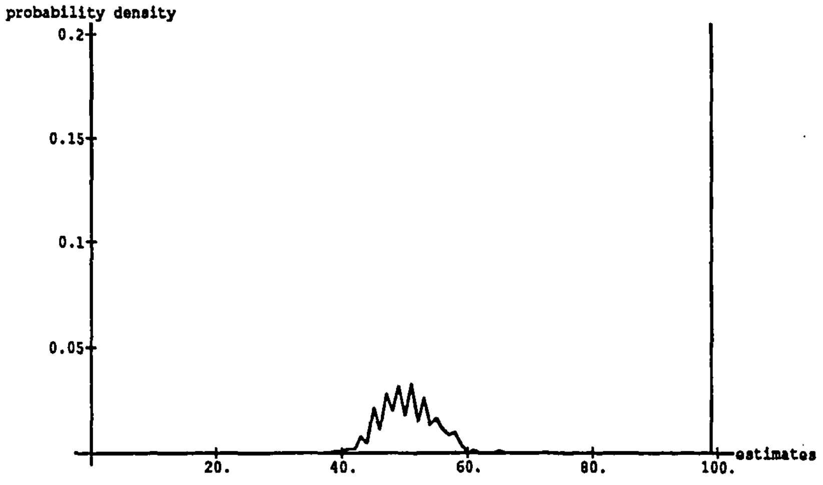


Fig. 6.7: Failure propagation estimates graph 4

Failure propagation analysis of Quadratic has also shown that the curve for $P_x$ appears to be fairly independent of both the perturbation function and input points. This observation is postulated after comparison of the $P_x$ curves from failure propagation graphs 1 and 4. The similarity of the curves is important since the infection interval range is unknown and the perturbation function parameters used may not have an infection interval distance of zero. Quadratic has also shown that a wide variety of combinations of faults of large fault size did not change the curves substantially between versions, with the exception being failure propagation graph 1, where a latter fault overcome the effects of perturbations at predecessor locations. Hence faults of smaller size only narrow the difference between such curves.

Infection graphs 1, 2, and 3(Figure 6.4, Figure 6.5, and Figure 6.6) for Quadratic contain similar results. All graphs received the same input points. Infection graph 1 was created by $P_{x-1}$ having the fault root2 := 2*a at location 7 and d := sqr(d) at location 5. Infection graph 2 was created by $P_{x-1}$ having the fault d := b at location 1 and rootexists := true at location 3. Infection graph 3 was created by $P_{x-1}$ having the fault if d < 5 at location 2 and root2 := -(b+d) at location 7.

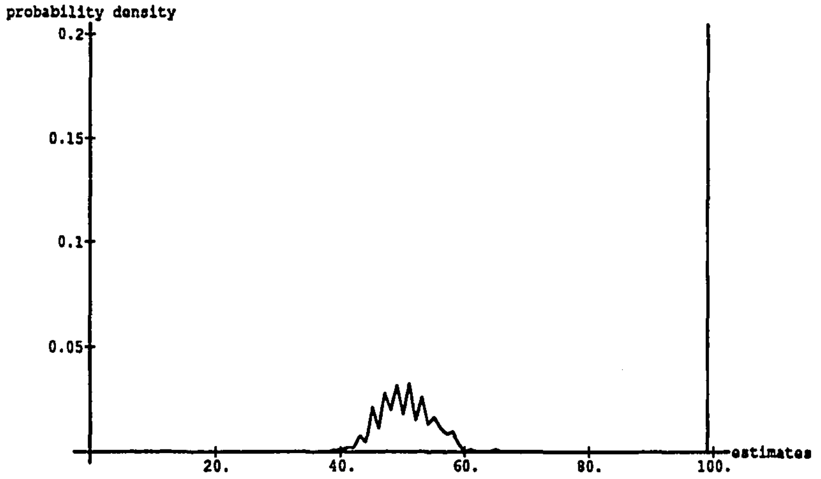


Fig. 6.8: Failure propagation estimates graph 5

## 6.4 Conclusions

Chapter 6 is not designed to show that propagation and infection analysis is the ultimate debugging tool, rather, that this particular implementation of PIA adequately corresponds to the propagation and infection analysis model, and additionally this model adequately approximates the behavior that real faults cause in programs. Debugging was a means of easily demonstrating this. This chapter has shown that propagation and infection analysis does indeed locate locations that more easily hide faults. In fact, propagation and infection analysis has good enough resolution to accurately quantify the impacts that a location has on the failure rate in order to use propagation and infection analysis for debugging. And remember that all of the analysis is done without an oracle. This means that incorrect programs are being used to debug themselves.

The programs used in Section 6.2, admittedly, are small and the analysis performed was incomplete. Complete propagation and infection analysis for an all but trivial program is all that could be done by this author without automation and faster machines. The results, however, do show that even partial propagation and infection analysis shows where faults can easily be hidden. This suggests that complete propagation and infection analysis will serve to give even better results for isolating such locations.

Since propagation and infection analysis is temporally expensive, it is desirable to begin propagation and infection analysis on the earliest version $P_k$ of $P$ such that versions $[P_{k+1}, P_{k+2}, ...]$ produce results similar to those for $P_k$. To begin propagation and infection analysis as soon as possible, the hypothesis as newer versions are released, the propagation and infection estimates of a preceding version change minimally for those of a successor version provided that the versions are syntactically[13] and semantically almost identical is introduced. Since faults of large size gave encouraging indicators in Section 6.3, faults of smaller size should produce results even more similar between versions.

The empirical conclusion of Chapter 6 is that a fault that is removed or inserted which does not drastically change the code structure will probably not change the propagation results discovered before fault removal or introduction. Thus two successive programs which are almost functionally equivalent yet structurally diverse will not have similar failure propagation estimates or infection estimates. Although $P_t$ probably contains faults, the faults' sizes are small and complete infection and failure propagation analysis need not be repeated for each change of $P_t$. This is important for models to which propagation and infection analysis's results will be applied.[14] Without this conclusion, full analysis would be required for each successive version. With this conclusion, there is a point in the software development cycle at which propagation and infection analysis may begin without having to reperform the analysis. Determination of exactly when this threshold point occurs is still under consideration, however this threshold point appears to occur when writing is completed and most debugging is completed as well.

[13] Two functionally equivalent programs may have totally different propagation and infection analysis results, hence syntactic restrictions are needed.

[14] For certain applications of propagation and infection analysis, this phenomenon makes the model more conservative; for other applications it may create problems.

# References

[1] BOEING COMPUTER SERVICES COMPANY SPACE AND MILITARY APPLICATIONS DIVISION. *Integrated Verification and Testing System IVTS System Maintenance Manual Volume V - Data-Flow Analysis.* August 1983.

[2] LEON J. OSTERWEIL AND LLOYD D. FOSDICK. Dave-a validation error detection and documentation system for fortran programs. *Software Practice and Experience*, pp. 473–486, October-December 1976.

[3] MICHAEL KARR AND DAVID B. LOVEMAN III. Incorporation of units into programming languages. *Communication of the ACM*, vol. 21:pp.385–391, May 1978.

[4] BODGAN KOREL. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9), September 1988.

[5] BODGAN KOREL. The program dependence graph in static program testing. *Information Processing Letters*, January 1987.

[6] LANUSZ W. LASKI AND BODGAN KOREL. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[7] C. W. BEDFORD AND E. E. HAMMOND JR. AND G. W. BEST AND J. R. LUX. *Fields and Functions, A Course in Precalculus Mathematics.* Macmillan Company, 1970.

[8] F. H. MARTIN. Hal/s - the avionics programming system for shuttle. *Proceedings AIAA Conference on Computers in Aerospace*, pp. 308–318, November 1977.

[9] JEFFREY M. VOAS AND LARRY J. MORELL. *Propagation Characteristics through Versions of One Program.* Technical Report WM-89-3, College of William and Mary, Department of Computer Science, October 1989.

[10] BRIAN W. KERNIGHAN AND P. J. PLAUGER. *The Elements of Programming Style*. McGraw-Hill Book Company, 1974.

[11] PETER W. PROTZEL. Automatically generated acceptance test: a software reliability experiment. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 196–203, July 1988.

[12] W. H. PRESS AND S. A. TEUKOLSKY AND B. P. FLANNERY AND W. T. VETTERLING. *Numerical Recipes The Art of Scientific Computing.* Cambridge University Press, 1986.

# Chapter 7

# Conclusions

## 7.1 Accomplishments

This thesis presents a structure-based methodology of quantifying the impact that a particular program location has on the program's output behavior; this structure-based methodology is both new and unique. It produces information never before available about a program's locations. And this information is produced without reliance upon a specification or oracle. Propagation and infection analysis is the model and implementation by which this role is quantified. The methodology combines aspects of both software testing and software verification techniques, however produces information not produced by either technique.

This thesis introduces the methodology for determining where faults can easily hide through the production of two statistically-derived estimates: the infection estimate and the propagation estimate. A low infection estimate indicates that the syntax was easily modified in such a manner that faults occurred and did not affect the values of variables in the data states during execution. A high infection estimate indicates that in general, changes to the syntax produced modified variable values in the data states during execution. A low propagation estimate indicates that modified variable values were put into data states which did not affect the output. A high propagation estimate indicates that modifications to values in the data states did affect the output behavior.

Abilities other than finding locations where faults can be easily hidden have been shown in the thesis. Included in these are the ability to partially detect missing paths or missing locations, and the ability to determine the minimal failure probability of the program relative to a class of faults or fault impacts. The thesis has also shown that the propagation and infection estimates can be used for analyzing software with a zero failure rate (ultra reliable software).

The thesis has shown that the code contains a wealth of information. It is a new white-box technique which provides information that no black-box analysis scheme can. Future applications of propagation and infection analysis estimates may be in certifying software. If it turns out that a particular piece of software contains more than some proportion of low failure propagation estimates, then the software could be rejected for rewriting, even if the software has never failed. The results from Chapter 6 are positive; they indicate that propagation and infection analysis or a methodology derived from it may be a tool which advances software quality.

## 7.2 Future Work

As a foundational model, the thesis generates more questions than answers. Areas for additional study beyond this thesis are not limited to but include:

**Perturbation Functions** The notion of a perturbation function simulating the impacts of an infinite number of faults is introduced. The simple step-wise algorithm for determining perturbation function parameters for two parameter perturbing distributions needs consideration. The notion of perturbing solely on the first data state may be shown to be inferior to perturbing uniformly over all occurrence data states. Additional work is needed into alternative methods for simulating the impacts on values during execution. Potentially actual faults should be used for creating viruses versus random functions.

**Artificial Data States Versus Natural Data States** The terminology introduced to describe the difference between these two types of data states is probably not the best, however the concepts they convey are important. If Method III value distribution creation is chosen, the question disappears assuming the location is eventually reached, because actual data states from an actual input point are available. However if sampled data states are created in some other fashion, then it is important that this question be resolved. It may turn out that Hypothesis 4.1 is not substantially threatened by using artificial data states, and if so, the cost involved in getting natural data states is not justified.

**Hierarchical Method for the Latent Failure Rate** This result currently suffers in usefulness by being a function of a particular path. Additional research is needed into determining how different are the latent failure rates of a set of paths from the latent failure rate of the path equivalence class after the necessary assumptions are made as to the number of iterations and which subpath is taken on which iteration. If evidence occurs that suggests that the assumptions can be made about the number of iterations as well as the subpaths taken without drastically altering the latent failure rate estimates between those of the paths and those of the generalizations of the path equivalence classes, then latent failure rates for path equivalence classes using the hierarchical method can be used which are not so conservative.

**Path Equivalence Class** The notion of a path equivalence class is used throughout the thesis as a means of partitioning the input space according to the following intuition: input points that follow the same path and reach the same locations will have a similar ability to reveal the same faults. Partitioning is useful for discussing the minimum failure probability for an execution. It is necessary to consider each location reached on an execution if an estimate of the minimum failure probability is attached to a sequence of reached locations. Whether the definitions of Chapter 2 are the best for satisfying this intuition is not known; work in this area is needed.

**Dispersion Histogram** The notion of the dispersion histogram is simple and straightforward. It stands to be shown that taking the results from Process Propagation Analyzer is the correct algorithm to tie the creation of the dispersion histogram to. Potentially the dispersion histogram should take into account the results from Process Infection Analyzer, because it may be that the perturbation functions used in Process Propagation Analyzer cause data state infections that normally do not occur. Hence these "artificial" infections are causing "artificial" pseudo-failures which would bias the dispersion histogram. The best scheme for both deriving the results and storing the results is still unresolved.

**Cancellation** Cancellation is possibly the most menacing problem in software testing. Both the viral and failure propagation estimates account for type I cancellation, however if there is to be any cost savings in the number of program executions required for propagation analysis, a static method for determining the situations under which cancellation occurs is important.

**When to Begin PIA** Chapter 6 has made an attempt to define certain properties that when true signal that propagation and infection analysis may begin without the penalty of reperforming propagation and infection analysis as the software changes. Potentially the restrictions on the input program of Chapter 5 are inaccurate, possibly too restrictive or not restrictive enough.

If machine speeds enable the performance of propagation and infection analysis to be considered as temporally trivial, this question vanishes. Currently this appears unlikely. More experimentation between successive versions is needed to verify or dispel Hypothesis 3.1.

**Infection Analysis Class of Faults** The definition of the fault class $\mathcal{F}$ is difficult. As the size of $\mathcal{F}$ increases, so do the number of semantic alternatives and thus the costs of performing infection analysis. Since the set of faults that could occur at a location is infinite, decreasing the size of this infinite set is essential. However if the definition of $\mathcal{F}$ becomes too restrictive, important classes of faults which frequently occur will be left out which will reduce the usefulness of infection analysis. If a listing of "common" faults is found relative to certain software applications, then such a listing could help define $\mathcal{F}$ for an input program of that type. The assumption has been made within $\mathcal{F}$ that each element is equally likely, however this assumption is not realistic. To be more realistic, $\mathcal{F}$ should be augmented by assigning probabilities for each element of $\mathcal{F}$. A study in the likelihood of fault classes is an entirely different problem which would be useful in defining $\mathcal{F}$.

**Algorithm Efficiency** From the pseudo-code shells provided for the Process Propagation Analyzer and Process Infection Analyzer, it is clear that except for trivially short programs, this methodology requires large amounts of execution time which with current technology requires either a parallel machine or a super-computer. However there are efficiencies possible from analytical schemes for both infection analysis and propagation analysis that will require fewer executions through the elimination of entire locations from needing dynamic analysis. As mentioned several times, parallelization of the computational algorithms appears to hold hope to this problem.

**Storing Data States by Method II** Finding a realistic scheme for Method II value distribution creation could lead to both advances in the speed of the algorithms of PIA and potentially a scheme for detecting "computer viruses." Internal data state storage is an important area for research.

**Automation** The ability to read in a program $P$ in a specific language, perform the preprocessing auxiliary processes which produce $P''$, and then input $P''$ into a process to create the pseudo-code shell for the two main processes is the overall goal of automating propagation and infection analysis. With such a scheme, any program in a particular language could have propagation and infection analysis immediately applied to it with virtually no manual requirements. The creation of an automated system to perform the PIA processes is a software-engineering project which hopefully will be undertaken.

Many of the unanswered questions and hypotheses can either be resolved or strongly implied by additional manually performed experiments on small sized input programs or having such a system. The availability of either a super-computer for these sequential algorithms or a parallel machine for a parallelized version of the algorithms is almost a mandatory requirement to currently use this methodology. Automation is meaningless if the pseudo-code shells take years to execute. Until more information is known about the pieces of the propagation and infection analysis methodology, automation should remain a goal for the future.

# Appendix A

# Mathematical Preliminaries

Appendix A briefly introduces four mathematical areas that are frequently assumed to be known throughout the seven chapters. These areas are: graphs, regular expressions, probability, and monte carlo simulation. Only the most basic rules from each of these areas is included in this appendix.

## A.1 Graph Terminology

A *graph* $G$ is an ordered pair $(V, E)$ made from a set of *edges* $E = \{e_{i_1}, e_{i_2}, e_{i_3}, ...\}$ and a set of *vertices* $V = \{v_1, v_2, v_3, ...\}$. An edge connects a pair of vertices $(a, b)$ where $a, b$ may be the same vertex. If they are the same vertex that edge is termed a *self-loop.* A graph is *finite* if $\mid V \mid$ is finite, otherwise it is *infinite* The vertices that a particular edge is incident to are called the *endpoints* of the edge.[2]

The *degree* of a vertex $v$, $d(v)$ is the number of times vertex $v$ occurs as an endpoint for the edges in $E$. A vertex whose degree is zero is *isolated.* The notation $u \overset{e}{-} v$ means that vertices $u$ and $v$ are connected by edge $e$ and thus $u$ and $v$ are adjacent.[2] A *path* is a sequence of edges$(e_{i_1}, e_{i_2}, ..., e_{i_j})$where:

1. $e_{i_k}$ and $e_{i_{k+1}}$ have a common endpoint for $k = 1..j - 1$;

2. if $e_{i_k}$ is not a self-loop and is not the first or last edge then $e_{i_k}$ shares one of its endpoints with $e_{i_{k-1}}$ and the other with $e_{i_{k+1}}$. [2]

A directed graph or *digraph* is a graph whose edges are ordered pairs; the first endpoint is the *head* and the second endpoint is the *tail.* The edge $(u \xrightarrow{e} v)e$ is directed from $u$ to $v$. A *directed path* is a sequence of edges such that the end-vertex of $e_{i_{k-1}}$ is the start-vertex of $e_{i_k}$. The *outdegree*, $d_{out}(u)$ of vertex $u$ is the number of edges which have $u$ as their start-vertex. The *indegree*, $d_{in}(u)$ of a vertex $u$ is the number of edges which have $u$ as their end-vertex.

A graph $G'$ is an ordered pair $(V', E')$ and is a *subgraph* of a graph G if $V' \subseteq V$ and $E' \subseteq E$. An arbitrary choice for $V'$ and $E'$ may not be a subgraph because they may not form a graph. [2]

## A.2 Regular Expression Terminology

The fundamental units from which structures are built is termed the *alphabet* which is a finite set of symbols. An example is the Roman alphabet {a,b,...,x}. A *string* over an alphabet is a finite sequence of symbols in the alphabet. A string may have no symbols at all; this string is a special string called the *empty string.* Any set of strings over an alphabet $\Sigma$ is called a *language.* A language can be specificed by listing all strings within it. [5, 1]

The *regular expressions* over an alphabet $\Sigma$ are the strings over the alphabet $\Sigma \cup \{), (, \vee, \cup, *\}$ such that the following hold:

- $\vee$ and each member if $\Sigma$ is a regular expression,
- If $\alpha$ and $\beta$ are regular expressions, then so is $(\alpha\beta)$,
- If $\alpha$ and $\beta$ are regular expressions, then so is $(\alpha \cup \beta)$,
- If $\alpha$ is a regular expression, then so is $\alpha^*$, and
- Nothing is a regular expression unless it follows from (1) through (4). [5]

## A.3 Conditional Probability and Independence Terminology

Let $E$ be an arbitrary event in a finite equiprobable sample space $S$ where the probability that $E$ occurs is greater than zero. And let $A$ be an arbitrary even in the same sample space. The probability that some event $A$ occurs given that $E$ has occurred termed the *conditional probability* of $A$ given $E$, denoted as $\Pr[A \mid E]$, and is defined as

$$\Pr[A \mid E] = \frac{\Pr[(A \cap E)]}{\Pr[E]}.$$

Then

$$\Pr[A \mid E] = \frac{\text{number of elements in } A \cap E}{\text{number of elements in} E}$$

And if the above equation is cross multiplied, the following formula known as the *multiplication theorem* is obtained[4]:

$$\Pr[(A \cap E)] = \Pr[E] \cdot \Pr[A \mid E]$$

Now assume that in the sample space $S$, there are $k$ mutually exclusive subsets, $S = B_1 \cup B_2 \cup ... \cup B_k$. Any subset $D$ of $S$ can be written then as $D = D \cap S = D \cap (B_1 \cup B_2 \cup ... \cup B_k)$ which is $(D \cap B_1) \cup (D \cap B_2) \cup ... \cup (D \cap B_k)$. So the probability of $D$ occurring is

$$\Pr[(D \cap B_1)] + \Pr[(D \cap B_2)] + ... + \Pr[(D \cap B_k)]$$

$$= \Pr[B_1] \cdot \Pr[D \mid B_1] + \Pr[B_2] \cdot \Pr[D \mid B_2] + \Pr[B_3] \cdot \Pr[D \mid B_3].$$

Additional references to conditional probabilities can be found in [7, 4].

## A.4 Monte Carlo Simulation Terminology

Using the notation in [6], let $x_1, x_2, .., x_n$ be a random sample where each $x_j$ is a Bernoulli random variable. Then the mean of this sample $m$ is

$$m = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and the standard deviation of the sample $s$ is

$$s = \sqrt{\sum_{i=1}^{n} x_i^2 - m^2} = \sqrt{m - m^2} = \sqrt{m(1-m)}.$$

From equation 4.11 in [3, 6], the interval defined by the two endpoints

$$m \pm \frac{t^* s}{\sqrt{n-1}}$$

is a $(1 - \alpha)\times$ 100% confidence interval estimate for the unknown true mean $\mu$. The parameter $(1-\alpha)$ is the *confidence level* associated with the interval estimate, and $t^*$ is the upper $1 - \alpha/2$ critical point for the $t$ distribution with $n - 1$ degrees of freedom. For a fixed sample size $n$ and $\alpha$,

$$t^* = t\_distribution(n-1, 1-\alpha/2).$$

In Monte Carlo simulation, the conventional confidence level is 95% [6]. When $n \to \infty$, $n \approx n-1$ and with $\alpha = 0.05$,

$$t^* = t\_distribution(n-1, 0.975) = 1.645 \approx 2.$$

Hence, the corresponding interval estimate of the sample mean is

$$m \pm 2\sqrt{\frac{m(1-m)}{n}}.$$

# References

[1] Daniel I. A. Cohen. *Introduction to Computer Theory.* John Wiley and Sons, Inc., 1986.

[2] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[3] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysus.* McGraw-Hill Book Company, 1982.

[4] Seymour Lipschutz. *Schaum's Outline Series Theory and Problems of Probability.* McGraw-Hill Book Company, 1965.

[5] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc., 1981.

[6] Steve Park. *Lecture notes on simulation, Unpvblished.* College of William and Mary, August 1988.

[7] William Mendenhall and Richard L. Scheaffer and Dennis D. Wackerly. *Mathematical Statistics with Applications.* Duxbury Press, 1981.

# Appendix B

# Programs for Propagation and Infection Estimates of qcksrt

```
 program  shell(input,output);
var
 ircounter :  array[1..32] of real;
 icounter :  array[1..22] of real;
 jcounter :  array[1..25] of real;
 fxcounter :  array[1..59] of real;
 arrcounter :  array[1..13] of real;
 seed : real;
 k,i : integer;
 counter1,counter2,counter3,counter4,counter5 : real;
 inputvalue : integer;
 sol,sol1,sol2,sol3,sol4,sol5 : integer; {solution from unperturbed versions}
 here1,here2,here3,here4,here5 : real;
 yes1,yes2,yes3,yes4,yes5 : boolean;
function random_park : real;
{steve park's random number generator}
const
 a = 16807.0;
 m = 2147483647.0;
 q = 127773.0;
 r = 2836;
var
 lo,hi,test : real;
begin
 hi := trunc(seed / q);
 lo := seed - q * hi;
 test := a * lo - r * hi;
  if test >  0  then
    seed := test
  else
    seed := test + m;
 random_park := seed / m;
end;
{——————discrete distributions——————————}
function equilikely(a,b : integer) : integer;
begin
equilikely := a + trunc((b-a+1) * random_park);
```

```
end;
{—————————continuous——————————}
function uniform(aa,bb:real) : real;
begin
 uniform := aa+((bb-aa) * random_park);
end;
{—————perturbation function——————————————}
function perturb_uni(x:real) : real;
var
newx : real;
counter : integer;
begin
newx := x;
 if (x=0.0)  then
    begin
      if random_park < 0.5  then
        newx:=1.0
       else
        newx:=-1.0
    end
else
 begin
 counter := 0;
  while (newx = x) do
    begin
     newx := uniform(0.5*x,1.5*x);
     counter := counter+1;
      if (newx=x)  and(counter=5)  then
        begin
         if random_park < 0.5  then
          newx:=x-1.0
         else
          newx:=x+1.0
        end;
    end;
 end;
perturb_uni := newx;
end;
function perturb_eqi(x:integer) : integer;
var newx : integer;
   counter : integer;
begin
newx := x;
 if x=0  then
    begin
      if random_park < 0.5  then
        newx:=1
       else
        newx:=-1;
    end
else
begin
counter:=0;
 while (newx = x) do
    begin
     newx := equilikely(trunc(x*0.999),trunc(1.001*x));
     counter := counter+1;
      if (newx=x)  and(counter=5)  then
        begin
         if random_park < 0.5  then
          newx:=x+1
         else
          newx:=x+1;
        end;
    end;

 end;
perturb_eqi := newx;
 end;
 function  test(val: integer): integer;
 const
   np = 11;
type
  gl array =  array [1..np] of integer;
 var
  i   :integer;
  errcnt : integer;
  seed   : real;
  start, finish : integer;
   procedure process(n: integer;  var arr: gl array);
  label
    11, 21, 22, 30, 99;
   const
    m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
   var
    l, jstack, j, ir, iq, i: integer;
    fx, fmi : real; a: integer;
    istack:  array[1..nstack] of integer;
   begin
    fmi := 1.0/fm;
    jstack := 0;
    l := 1;
    ir := n;
    fx := 0.0;
   while true  do  begin
    if (ir-l) <  m  then  begin
  (* < ============== 1*)
       for j := l+1  to ir  do  begin
   a := arr[j];
   for i := j-1 down to 1  do  begin
       if (arr[i] < = a)  then  goto 11;
      arr[i+1] := arr[i];
   end;
  i := 0;
  11:     arr[i+1] := a
       end;
       if (jstack = 0)  then  goto 99;
      ir := istack[jstack];
ir := istack[jstack];  ack-1];
l := istack[jstack-1];ck-2;
     end
     else  begin
       i := l;
       j := ir;
       fx := (fx*fa+fc)/fm;
  (* < ============== *)
         iq := l+(ir-l+1)* trunc(fx*fmi);
      a := arr[iq];
      arr[iq] := arr[l];
  21:  if (j > 0)  then  begin
         if (a <  arr[j])  then  begin
    j := j-1;
```

```
       goto 21;
          end
         end;
         if (j < = i) then begin
           arr[i] := a;
(* < ==================== *)
           goto 30;
         end;
        arr[i] := arr[j];
        i := i + 1;
  22:     if (i < = n) then
           if (a > arr[i]) then begin
        i := i + 1;
          goto 22;
    end;
         if (j < = i) then begin
           arr[j] := a;
           i := j;
            goto 30;
         end;
        arr[j] := arr[i];
        j := j - l;
(* < ==================== *)
         goto 21;
  30: jstack := jstack + 2;
         if (jstack > nstack) then begin
  writeln('Overflow');
           end;
         if (ir -i) > = (i-l) then begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
  ir := i -1;
(* < ==================== *)
           end
           else begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
           end
       end
      end;
99: end; (* procedure process *)
 function anonymous(val : integer): integer;
 var
   arr   : gl array;
   n     : integer;
   i     : integer;
   count :  array[0..7] of integer;
   correct :  array[0..11] of integer;
   pos,k   : integer;
   error : boolean;
begin (* anonymous *)
    i:= 1;
     while val < > 0 do begin
       arr[i] := val mod 8;
       val := val  div 8;
       i := i + 1;
     end;
    n := i -1;
     while i < = np do begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
  val := 0;
    for i := np down to 1  do  begin
      val := val * 8 + arr[i]
    end;
  anonymous := val;
end;  (* anonymous *)
begin
  test := anonymous(val);
end;

function  test1(val: integer): integer;
const
  np = 11;
type
  gl array =  array [1..np] of integer;
var
  i    :integer;
  errcnt : integer;
  seed   : real;
  start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]   of integer;
    begin
     fmi := 1.0/fm;
     jstack := 0;
     l := 1;
     ir := n;
     fx := 0.0;
    while true  do  begin
     if ((ir-l) <  m) < >  ((ir-l) >  m)  then ircounter[1] := ircounter[1] + 1;
     if ((ir-l) <  m) < >  ((l-ir) <  m)  then ircounter[2] := ircounter[2] + 1;
     if ((ir-l) <  m) < >  ((ir-l) = m)  then ircounter[3] := ircounter[3] + 1;
     if ((ir-l) <  m) < >  ((ir-l) < = m)  then ircounter[4] := ircounter[4] + 1;
     if ((ir-l) <  m) < >  ((ir-l) > = m)  then ircounter[5] := ircounter[5] + 1;
     if ((ir-l) <  m) < >  ((ir-l) <  fm)  then ircounter[6] := ircounter[6] + 1;
     if ((ir-l) <  m) < >  ((ir-l) <  fmi)  then ircounter[7] := ircounter[7] + 1;
     if ((ir-l) <  m) < >  ((ir) <  m)  then ircounter[8] := ircounter[8] + 1;
     if ((ir-l) <  m) < >  ((l) <  m)  then ircounter[9] := ircounter[9] + 1;
     if ((ir-l) <  m) < >  ((-l) <  m)  then ircounter[10] := ircounter[10] + 1;
     if ((ir-l) <  m) < >  ((ir+l) <  m)  then ircounter[11] := ircounter[11] + 1;
     if ((ir-l) <  m) < >  ((ir*l) <  m)  then ircounter[12] := ircounter[12] + 1;
     if ((ir-l) <  m) < >  ((ir-1) <  m)  then ircounter[13] := ircounter[13] + 1;
     if ((ir-l) <  m) < >  ((jstack-l) <  m)  then ircounter[14] := ircounter[14] + 1;
     if ((ir-l) <  m) < >  ((istack[jstack]-l) <  m)  then ircounter[15] := ircounter[15] + 1;
     if ((ir-l) <  m) < >  ((arr[i]-l) <  m)  then ircounter[16] := ircounter[16] + 1;
     if ((ir-l) <  m) < >  ((arr[i+1]-l) <  m)  then ircounter[17] := ircounter[17] + 1;
     if ((ir-l) <  m) < >  ((ir-jstack) <  m)  then ircounter[18] := ircounter[18] + 1;
     if ((ir-l) <  m) < >  ((ir-fmi) <  m)  then ircounter[19] := ircounter[19] + 1;
     if ((ir-l) <  m) < >  ((ir-arr[i]) <  m)  then ircounter[20] := ircounter[20] + 1;
     if ((ir-l) <  m) < >  ((ir-arr[i+1]) <  m)  then ircounter[21] := ircounter[21] + 1;
     if ((ir-l) <  m) < >  ((ir-istack[jstack]) <  m)  then ircounter[22] := ircounter[22] + 1;
  if ((ir-l) < m) < > ((ir-l) < jstack) then ircounter[23] := ircounter[23] + 1;
  if ((ir-l) < m) < > ((ir-l) < arr[i]) then ircounter[24] := ircounter[24] + 1;
  if ((ir-l) < m) < > ((ir-l) < arr[i+1]) then ircounter[25] := ircounter[25] + 1;
  if ((ir-l) < m) < > ((ir-l) < istack[jstack]) then ircounter[26] := ircounter[26] + 1;
  if ((ir-l) < m) < > ((1-l) < m) then ircounter[27] := ircounter[27] + 1;
  if ((ir-l) < m) < > ((ir-a) < m) then ircounter[28] := ircounter[28] + 1;
  if ((ir-l) < m) < > ((ir-l) < a) then ircounter[29] := ircounter[29] + 1;
  if ((ir-l) < m) < > ((ir-l) < n) then ircounter[30] := ircounter[30] + 1;
  if ((ir-l) < m) < > ((ir-n) < m) then ircounter[31] := ircounter[31] + 1;
  if ((ir-l) < m) < > ((n-l) < m) then ircounter[32] := ircounter[32] + 1;
 counter1 := counter1 + 1;
  if (ir-l) < m then begin
    for j := l+1 to ir do begin
a := arr[j];
 for i := j-1 down to 1 do begin
    if (arr[i] < = a) then goto 11;
   arr[i+1] := arr[i];
 end;
i := 0;
11:     arr[i+1] := a
    end;
    if (jstack = 0) then goto 99;
   ir := istack[jstack];
   l := istack[jstack-1];
   jstack := jstack-2;
  end
  else begin
   i := l;
   j := ir;
   fx := (fx*fa+fc)/fm;
 (* < ============2*)
     iq := l+(ir-l+1)* trunc(fx*fmi);
   a := arr[iq];
   arr[iq] := arr[l];
21:  if (j >  0) then begin
       if (a <  arr[j]) then begin
  j := j-1;
   goto 21;
       end
    end;
    if (j < = i) then begin
      arr[i] := a;
(* < ========== *)
      goto 30;
    end;
   arr[i] := arr[j];
   i := i + 1;
22:    if (i < = n) then
      if (a > arr[i]) then begin
    i := i + 1;
     goto 22;
 end;
    if (j < = i) then begin
      arr[j] := a;
      i := j;
       goto 30;
    end;
   arr[j] := arr[i];
   j := j - l;
(* < ========== *)
    goto 21;
  30: jstack := jstack + 2;
        if (jstack > nstack) then begin
  writeln('Overflow');
        end;
        if (ir -i) > = (i-l) then begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
  ir := i -1;
 (* < ========== *)
        end
        else begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
        end
     end
    end;
99: end; (* procedure process *)
function anonymous(val : integer): integer;
var
  arr   : gl array;
  n     : integer;
  i     : integer;
  count : array[0..7]  of integer;
  correct : array[0..11]  of integer;
  pos,k   : integer;
  error : boolean;
begin (* anonymous *)
   i:= 1;
    while val < > 0 do begin
      arr[i] := val  mod 8;
      val := val  div 8;
      i := i + 1;
    end;
   n := i -1;
    while i < = np do begin
      arr[i] := 0;
      i := i + 1;
    end;
   process(n,arr);
  val := 0;
   for i := np down to 1 do begin
     val := val * 8 + arr[i]
   end;
  anonymous := val;
end; (* anonymous *)
begin
  test1 := anonymous(val);
end;


function test2(val: integer): integer;
const
  np = 11;
type
  gl array = array [1..np]  of integer;
var
  i    :integer;
  errcnt : integer;
  seed   : real;
  start, finish : integer;
```

```
procedure process(n: integer;  var arr: gl array);
 label
  11, 21, 22, 30, 99;
 const
  m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
 var
  l, jstack, j, ir, iq, i: integer;
  fx, fmi : real; a: integer;
  istack:  array[1..nstack]   of integer;
 begin
  fmi := 1.0/fm;
  jstack := 0;
  l := 1;
  ir := n;
  fx := 0.0;
  while true  do  begin
  if (ir-l) <  m  then  begin
(* < ========== *)
    for j := l+1  to ir  do  begin
a := arr[j];
 for i := j-1 down to 1  do  begin
    if (arr[i] < = a)  then  goto 11;
   arr[i+1] := arr[i];
 end;
i := 0;
11:    arr[i+1] := a
    end;
    if (jstack = 0)  then  goto 99;
   ir := istack[jstack];
   l := istack[jstack-1];
   jstack := jstack-2;
  end
  else  begin
   i := l;
   j := ir;
    if (fx*fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[1] := fxcounter[1]+1;
    if (fa*fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[2] := fxcounter[2]+1;
    if (fa-fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[3] := fxcounter[3]+1;
    if (5*fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[4] := fxcounter[4]+1;
    if (fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[5] := fxcounter[5]+1;
    if (fm*fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[6] := fxcounter[6]+1;
    if (fx/fa-fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[7] := fxcounter[7]+1;
    if (fx*fa*fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[8] := fxcounter[8]+1;
    if (fx*fa*fc)+fm < >  (fx*fa+fc)/fm  then fxcounter[9] := fxcounter[9]+1;
    if (fx*fa*fc)*fm < >  (fx*fa+fc)/fm  then fxcounter[10] := fxcounter[10]+1;
    if fm/(fx*fa+fc) < >  (fx*fa+fc)/fm  then fxcounter[11] := fxcounter[11]+1;
    if (fx*fa+fc)*fx < >  (fx*fa+fc)/fm  then fxcounter[12] := fxcounter[12]+1;
    if (fx*fa+fc)*fa < >  (fx*fa+fc)/fm  then fxcounter[13] := fxcounter[13]+1;
    if (fx*fa+fc)*fc < >  (fx*fa+fc)/fm  then fxcounter[14] := fxcounter[14]+1;
    if (i*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[15] := fxcounter[15]+1;
    if (fx*i+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[16] := fxcounter[16]+1;
    if (fx*fa+i)/fm < >  (fx*fa+fc)/fm  then fxcounter[17] := fxcounter[17]+1;
    if (fx*fa+fc)/i < >  (fx*fa+fc)/fm  then fxcounter[18] := fxcounter[18]+1;
    if (l*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[19] := fxcounter[19]+1;
    if (fx*l+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[20] := fxcounter[20]+1;
    if (fx*fa+l)/fm < >  (fx*fa+fc)/fm  then fxcounter[21] := fxcounter[21]+1;
    if (fx*fa+fc)/l < >  (fx*fa+fc)/fm  then fxcounter[22] := fxcounter[22]+1;
    if (j*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[23] := fxcounter[23]+1;
    if (fx*j+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[24] := fxcounter[24]+1;
    if (fx*fa+j)/fm < >  (fx*fa+fc)/fm  then fxcounter[25] := fxcounter[25]+1;
    if (fx*fa+fc)/j < >  (fx*fa+fc)/fm  then fxcounter[26] := fxcounter[26]+1;
```

```
      if (ir*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[27] := fxcounter[27]+1;
      if (fx*ir+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[28] := fxcounter[28]+1;
      if (fx*fa+ir)/fm < >  (fx*fa+fc)/fm  then fxcounter[29] := fxcounter[29]+1;
      if (fx*fa+fc)/ir < >  (fx*fa+fc)/fm  then fxcounter[30] := fxcounter[30]+1;
      if (fx*fx+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[31] := fxcounter[31]+1;
      if (fx*fa+fx)/fm < >  (fx*fa+fc)/fm  then fxcounter[32] := fxcounter[32]+1;
      if (fx*fa+fc)/fx < >  (fx*fa+fc)/fm  then fxcounter[33] := fxcounter[33]+1;
      if (fa*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[34] := fxcounter[34]+1;
      if (fx*fa+fa)/fm < >  (fx*fa+fc)/fm  then fxcounter[35] := fxcounter[35]+1;
      if (fx*fa+fc)/fa < >  (fx*fa+fc)/fm  then fxcounter[36] := fxcounter[36]+1;
      if (fc*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[37] := fxcounter[37]+1;
      if (fx*fc+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[38] := fxcounter[38]+1;
      if (fx*fa+fc)/fc < >  (fx*fa+fc)/fm  then fxcounter[39] := fxcounter[39]+1;
      if (fm*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[40] := fxcounter[40]+1;
      if (fx*fm+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[41] := fxcounter[41]+1;
      if (fx*fa+fm)/fm < >  (fx*fa+fc)/fm  then fxcounter[42] := fxcounter[42]+1;
      if (iq*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[43] := fxcounter[43]+1;
      if (fx*iq+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[44] := fxcounter[44]+1;
      if (fx*fa+iq)/fm < >  (fx*fa+fc)/fm  then fxcounter[45] := fxcounter[45]+1;
      if (fx*fa+fc)/iq < >  (fx*fa+fc)/fm  then fxcounter[46] := fxcounter[46]+1;
      if (fmi*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[47] := fxcounter[47]+1;
      if (fx*fmi+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[48] := fxcounter[49]+1;
      if (fx*fa+fmi)/fm < >  (fx*fa+fc)/fm  then fxcounter[50] := fxcounter[50]+1;
      if (fx*fa+fc)/fmi < >  (fx*fa+fc)/fm  then fxcounter[51] := fxcounter[51]+1;
      if (a*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[52] := fxcounter[52]+1;
      if (fx*a+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[53] := fxcounter[53]+1;
      if (fx*fa+a)/fm < >  (fx*fa+fc)/fm  then fxcounter[54] := fxcounter[54]+1;
      if (fx*fa+fc)/a < >  (fx*fa+fc)/fm  then fxcounter[55] := fxcounter[55]+1;
      if (arr[iq]*fa+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[56] := fxcounter[56]+1;
      if (fx*arr[iq]+fc)/fm < >  (fx*fa+fc)/fm  then fxcounter[57] := fxcounter[57]+1;
      if (fx*fa+arr[iq])/fm < >  (fx*fa+fc)/fm  then fxcounter[58] := fxcounter[58]+1;
      if (fx*fa+fc)/arr[iq] < >  (fx*fa+fc)/fm  then fxcounter[59] := fxcounter[59]+1;
     fx := (fx*fa+fc)/fm;
     counter2 := counter2 + 1;
       iq := l+(ir-l+1)* trunc(fx*fmi);
     a := arr[iq];
     arr[iq] := arr[l];
21:  if (j > 0) then begin
       if (a < arr[j]) then begin
   j := j-1;
    goto 21;
       end
      end;
     if (j < = i) then begin
       arr[i] := a;
(* < ========== *)
       goto 30;
      end;
     arr[i] := arr[j];
     i := i + 1;
22:    if (i < = n) then
       if (a > arr[i]) then begin
     i := i + 1;
       goto 22;
  end;
     if (j < = i) then begin
       arr[j] := a;
       i := j;
       goto 30;
      end;
     arr[j] := arr[i];
```

```
      j := j - l;
  (* < ========== *)
       goto 21;
  30: jstack := jstack + 2;
       if (jstack > nstack) then begin
   writeln('Overflow');
        end;
        if (ir -i) > = (i-l) then begin
   istack[jstack] := ir;
   istack[jstack-1] := i + 1;
   ir := i -1;
  (* < ========== *)
        end
        else begin
   istack[jstack] := i-1;
   istack[jstack-1] := l;
   l := i+1;
        end
     end
    end;
99: end; (* procedure process *)
 function anonymous(val : integer): integer;
 var
   arr   : gl array;
   n     : integer;
   i     : integer;
   count : array[0..7] of integer;
   correct : array[0..11] of integer;
   pos,k  : integer;
   error : boolean;
 begin (* anonymous *)
    i:= 1;
      while val < > 0 do begin
       arr[i] := val mod 8;
       val := val div 8;
       i := i + 1;
     end;
    n := i -1;
      while i < = np do begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
   val := 0;
    for i := np down to 1 do begin
      val := val * 8 + arr[i]
    end;
   anonymous := val;
 end; (* anonymous *)
 begin
   test2 := anonymous(val);
 end;

 function test3(val: integer): integer;
 const
   np = 11;
type
   gl array = array [1..np] of integer;
 var
   i   :integer;
   errcnt : integer;
```

```
seed  : real;
start, finish : integer;
 procedure process(n: integer;  var arr: gl array);
  label
   11, 21, 22, 30, 99;
  const
   m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
  var
   l, jstack, j, ir, iq, i: integer;
   fx, fmi : real; a: integer;
   istack:  array[1..nstack]  of integer;
  begin
   fmi := 1.0/fm;
   jstack := 0;
   l := 1;
   ir := n;
   fx := 0.0;
   while true  do  begin
   if (ir-l) < m  then  begin
(* < ============ *)
     for j := l+1  to ir  do  begin
 a := arr[j];
 for i := j-1 down to 1  do  begin
    if (arr[i] < = a)  then  goto 11;
   arr[i+1] := arr[i];
  end;
i := 0;
11:    arr[i+1] := a
     end;
     if (jstack = 0)  then  goto 99;
    ir := istack[jstack];
    l := istack[jstack-1];
    jstack := jstack-2;
   end
   else  begin
    i := l;
    j := ir;
    fx := (fx*fa+fc)/fm;
(* < ============ *)
      iq := l+(ir-l+1)* trunc(fx*fmi);
    a := arr[iq];
    arr[iq] := arr[l];
21:  if (j > 0)  then  begin
       if (a <  arr[j])  then  begin
 j := j-1;
  goto 21;
       end
     end;
     if (j < = i)  then  begin
       if a < >  1  then  arrcounter[1] := arrcounter[1] + 1;
       if a < >  a-1  then  arrcounter[2] := arrcounter[2] + 1;
       if a < >  i  then  arrcounter[3] := arrcounter[3] + 1;
       if a < >  j  then  arrcounter[4] := arrcounter[4] + 1;
       if a < >  1-a  then  arrcounter[5] := arrcounter[5] + 1;
       if a < >  5*a + 1  then  arrcounter[6] := arrcounter[6] + 1;
       if a < >  arr[iq]  then  arrcounter[7] := arrcounter[7] + 1;
       if a < >  iq  then  arrcounter[8] := arrcounter[8] + 1;
       if a < >  arr[l]  then  arrcounter[9] := arrcounter[9] + 1;
       if a < >  l  then  arrcounter[10] := arrcounter[10] + 1;
       if a < >  arr[j]  then  arrcounter[11] := arrcounter[11] + 1;
       if a < >  n  then  arrcounter[12] := arrcounter[12] + 1;
```

```
          if a < > fx then arrcounter[13] := arrcounter[13] + 1;
         arr[i] := a;
         counter3 := counter3 + 1;
          goto 30;
       end;
      arr[i] := arr[j];
      i := i + 1;
  22:       if (i < = n) then
          if (a > arr[i]) then begin
       i := i + 1;
        goto 22;
    end;
       if (j < = i) then begin
         arr[j] := a;
         i := j;
          goto 30;
       end;
      arr[j] := arr[i];
      j := j - l;
  (* < ========== *)
       goto 21;
  30: jstack := jstack + 2;
       if (jstack > nstack) then begin
  writeln('Overflow');
        end;
        if (ir -i) > = (i-l) then begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
  ir := i -1;
  (* < ========== *)
        end
        else begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
        end
     end
     end;
99: end; (* procedure process *)
 function anonymous(val : integer): integer;
 var
  arr   : gl array;
  n     : integer;
  i     : integer;
  count : array[0..7] of integer;
  correct : array[0..11] of integer;
  pos,k  : integer;
  error : boolean;
 begin (* anonymous *)
    i:= 1;
      while val < > 0 do begin
       arr[i] := val mod 8;
       val := val div 8;
       i := i + 1;
     end;
    n := i -1;
      while i < = np do begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
    val := 0;
     for i := np down to 1  do  begin
       val := val * 8 + arr[i]
     end;
    anonymous := val;
  end;  (* anonymous *)
  begin
    test3 := anonymous(val);
  end;


  function  test4(val: integer): integer;
  const
    np = 11;
 type
   gl array =  array [1..np]   of integer;
  var
    i    :integer;
    errcnt : integer;
    seed   : real;
    start, finish : integer;
     procedure process(n: integer;  var arr: gl array);
      label
       11, 21, 22, 30, 99;
     const
       m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
     var
       l, jstack, j, ir, iq, i: integer;
       fx, fmi : real; a: integer;
       istack:  array[1..nstack]   of integer;
     begin
       fmi := 1.0/fm;
       jstack := 0;
       l := 1;
       ir := n;
       fx := 0.0;
      while true  do  begin
      if (ir-l) < m  then  begin
   (* < ============ *)
        for j := l+1  to ir  do  begin
    a := arr[j];
     for i := j-1 down to 1  do  begin
        if (arr[i] < = a)  then  goto 11;
       arr[i+1] := arr[i];
     end;
    i := 0;
    11:    arr[i+1] := a
ir := istack[jstack];
l := istack[jstack-1]; 0)  then  goto 99;
        ir := istack[jstack];
        l := istack[jstack-1];
        jstack := jstack-2;
      end
      else  begin
        i := l;
        j := ir;
        fx := (fx*fa+fc)/fm;
   (* < ============ *)
          iq := l+(ir-l+1)* trunc(fx*fmi);
        a := arr[iq];
        arr[iq] := arr[l];
    21:  if (j > 0)  then  begin
```

```
      if (a < arr[j]) then begin
   j := j-1;
   goto 21;
      end
    end;
    if (j < = i) then begin
     arr[i] := a;
(* < ============ *)
      goto 30;
    end;
   arr[i] := arr[j];
   i := i + 1;
22:   if (i < = n) then
      if (a > arr[i]) then begin
   i := i + 1;
    goto 22;
 end;
    if (j < = i) then begin
     arr[j] := a;
     i := j;
      goto 30;
    end;
   arr[j] := arr[l];
    if j-l < > j-1 then jcounter[1] := jcounter[1] + 1;
    if j-l < > j+1 then jcounter[2] := jcounter[2] + 1;
    if j-l < > j+l then jcounter[3] := jcounter[3] + 1;
    if j-l < > l-j then jcounter[4] := jcounter[4] + 1;
    if j-l < > j*l then jcounter[5] := jcounter[5] + 1;
    if j-l < > j  div l then jcounter[6] := jcounter[6] + 1;
    if j-l < > l then jcounter[7] := jcounter[7] + 1;
    if j-l < > 1 then jcounter[8] := jcounter[8] + 1;
    if j-l < > j then jcounter[9] := jcounter[9] + 1;
    if j-l < > j-j then jcounter[10] := jcounter[10] + 1;
    if j-l < > 5*j - 3*l then jcounter[11] := jcounter[11] + 1;
    if j-l < > j*j-2*l-3 then jcounter[12] := jcounter[12] + 1;
    if j-l < > j-arr[i] then jcounter[13] := jcounter[14] + 1;
    if j-l < > j-i then jcounter[15] := jcounter[15] + 1;
    if j-l < > j-ir then jcounter[16] := jcounter[16] + 1;
    if j-l < > j-istack[jstack] then jcounter[17] := jcounter[17] + 1;
    if j-l < > j-istack[jstack-1]  then jcounter[18] := jcounter[18] + 1;
    if j-l < > arr[i]-l then jcounter[19] := jcounter[19] + 1;
    if j-l < > i-l then jcounter[20] := jcounter[20] + 1;
    if j-l < > ir-l then jcounter[21] := jcounter[21] + 1;
    if j-l < > istack[jstack]-l then jcounter[22] := jcounter[23] + 1;
    if j-l < > istack[jstack-1] -l then jcounter[24] := jcounter[24] + 1;
    if j-l < > l-l then jcounter[25] := jcounter[25] + 1;
   j := j - l;
   counter4 := counter4 + 1;
    goto 21;
30: jstack := jstack + 2;
     if (jstack > nstack) then begin
writeln('Overflow');
     end;
     if (ir -i) > = (i-l) then begin
istack[jstack] := ir;
istack[jstack-1] := i + 1;
ir := i -1;
(* < ============ *)
     end
     else begin
istack[jstack] := i-1;
```

```
   istack[jstack-1] := l;
   l := i+1;
         end
      end
      end;
99: end; (*  procedure process *)
 function anonymous(val : integer): integer;
 var
   arr   : gl array;
   n     : integer;
   i     : integer;
   count : array[0..7]  of integer;
   correct : array[0..11]  of integer;
   pos,k   : integer;
   error : boolean;
 begin (* anonymous *)
    i:= 1;
      while val < >  0  do  begin
       arr[i] := val   mod 8;
       val := val   div 8;
       i := i + 1;
     end;
    n := i -1;
      while i < = np  do  begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);

   val := 0;
    for i := np down to 1  do  begin
      val := val * 8 + arr[i]
    end;
   anonymous := val;
 end;  (* anonymous *)
 begin
    test4 := anonymous(val);
 end;


 function  test5(val: integer): integer;
 const
   np = 11;
type
   gl array =  array [1..np]   of integer;
 var
   i    :integer;
   errcnt : integer;
   seed   : real;
   start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]   of integer;
    begin
     fmi := 1.0/fm;
```

```
  jstack := 0;
  l := 1;
  ir := n;
  fx := 0.0;
  while true do begin
  if (ir-l) < m then begin
(* < ============ *)
    for j := l+1 to ir do begin
 a := arr[j];
 for i := j-1 down to 1 do begin
    if (arr[i] < = a) then goto 11;
   arr[i+1] := arr[i];
 end;
i := 0;
11:    arr[i+1] := a
    end;
    if (jstack = 0) then goto 99;
   ir := istack[jstack];
   l := istack[jstack-1];
   jstack := jstack-2;
  end
  else begin
   i := l;
   j := ir;
   fx := (fx*fa+fc)/fm;
(* < ============ *)
     iq := l+(ir-l+1)* trunc(fx*fmi);
   a := arr[iq];
   arr[iq] := arr[l];
21: if (j > 0) then begin
      if (a < arr[j]) then begin
 j := j-1;
  goto 21;
      end
    end;
    if (j < = i) then begin
     arr[i] := a;
(* < ============ *)
      goto 30;
    end;
   arr[i] := arr[j];
   i := i + 1;
22:    if (i < = n) then
      if (a > arr[i]) then begin
    i := i + 1;
     goto 22;
 end;
    if (j < = i) then begin
     arr[j] := a;
     i := j;
      goto 30;
    end;
   arr[j] := arr[i];
   j := j - l;
(* < ============ *)
    goto 21;
 30: jstack := jstack + 2;
    if (jstack > nstack) then begin
 writeln('Overflow');
     end;
     if (ir -i) > = (i-l) then begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
      if i-1 < > i  then icounter[1] := icounter[1] + 1;
      if i-1 < > i+1  then icounter[2] := icounter[2] + 1;
      if i-1 < > i+ir  then icounter[3] := icounter[3] + 1;
      if i-1 < > ir-i  then icounter[4] := icounter[4] + 1;
      if i-1 < > i*ir  then icounter[5] := icounter[5] + 1;
      if i-1 < > i   div ir  then icounter[6] := icounter[6] + 1;
      if i-1 < > ir  then icounter[7] := icounter[7] + 1;
      if i-1 < > 1  then icounter[8] := icounter[8] + 1;
      if i-1 < > i  then icounter[9] := icounter[9] + 1;
      if i-1 < > i-i  then icounter[10] := icounter[10] + 1;
      if i-1 < > i*i-i-1  then icounter[11] := icounter[11] + 1;
      if i-1 < > i+i  then icounter[12] := icounter[12] + 1;
      if i-1 < > i-jstack  then icounter[13] := icounter[13] + 1;
      if i-1 < > i-arr[i]  then icounter[14] := icounter[14] + 1;
      if i-1 < > i-a  then icounter[15] := icounter[15] + 1;
      if i-1 < > i-n  then icounter[16] := icounter[16] + 1;
      if i-1 < > i-ir  then icounter[17] := icounter[17] + 1;
      if i-1 < > jstack-1  then icounter[18] := icounter[18] + 1;
      if i-1 < > arr[i]-1  then icounter[19] := icounter[19] + 1;
      if i-1 < > a-1  then icounter[20] := icounter[20] + 1;
      if i-1 < > n-1  then icounter[21] := icounter[21] + 1;
      if i-1 < > ir-1  then icounter[22] := icounter[22] + 1;
     ir := i -1;
     counter5 := counter5 + 1;
       end
       else  begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
       end
     end
     end;
99: end; (*  procedure process *)
 function anonymous(val : integer): integer;
 var
  arr   : gl array;
  n     : integer;
  i     : integer;
  count :  array[0..7]   of integer;
  correct :  array[0..11]   of integer;
  pos,k   : integer;
  error : boolean;
 begin (* anonymous *)
  i:= 1;
    while val < >  0  do  begin
     arr[i] := val   mod 8;
     val := val   div 8;
     i := i + 1;
    end;
  n := i -1;
    while i < = np  do  begin
     arr[i] := 0;
     i := i + 1;
    end;
   process(n,arr);
  val := 0;
   for i := np down to 1  do  begin
    val := val * 8 + arr[i]
   end;
   anonymous := val;
 end; (* anonymous *)
 begin
   test5 := anonymous(val);
 end;


 begin(*************main***********************************)
seed := 123456722.0;
counter1:=0.0;
counter2:=0.0;
counter3:=0.0;
counter4:=0.0;
counter5:=0.0;
 for k := 1  to 32 do
  ircounter[k] := 0.0;
 for k := 1  to 22 do
  icounter[k] := 0.0;
 for k := 1  to 25 do
   jcounter[k] := 0.0;
 for k := 1  to 59  do
  fxcounter[k] := 0.0;
 for k := 1  to 13 do
  arrcounter[k] := 0.0;
 for k := 1  to 100000 do
   begin
     inputvalue := equilikely(0,2400000);
     sol :=  test(inputvalue);
     sol1 := test1(inputvalue);
     sol2 := test2(inputvalue);
     sol3 := test3(inputvalue);
     sol4 := test4(inputvalue);
     sol5 := test5(inputvalue);
   end;
{output statements here}
 end.
 program shell(input,output);
 var
  seed : real;
  k,i : integer;
  counter1,counter2,counter3,counter4,counter5 : real;
  inputvalue : integer;
  sol,sol1,sol2,sol3,sol4,sol5 : integer; {solution from unperturbed versions}
  here1,here2,here3,here4,here5 : real;
  yes1,yes2,yes3,yes4,yes5 : boolean;
  perturb : array[1..5] of boolean;
 function random_park : real;
{steve park's random number generator}
 const
  a = 16807.0;
  m = 2147483647.0;
  q = 127773.0;
  r = 2836;
 var
   lo,hi,test : real;
 begin
  hi := trunc(seed / q);
  lo := seed - q * hi;
  test := a * lo - r * hi;
   if test > 0 then
     seed := test
   else
     seed := test + m;
  random_park := seed / m;
 end;
{——————————discrete distributions——————————————————}
 function equilikely(a,b : integer) : integer;
 begin
equilikely := a + trunc((b-a+1) * random_park);
 end;
{————————————————continuous——————————————————}
 function uniform(aa,bb:real) : real;
 begin
  uniform := aa+((bb-aa) * random_park);
 end;
{—————————perturbation function————————————————————————}
 function perturb_uni(x:real) : real;
 var
  newx : real;
  counter : integer;
 begin
newx := x;
 if (x=0.0) then
    begin
      if random_park < 0.5 then
        newx:=random_park
       else
        newx:=-random_park
    end
 else
  begin
  counter := 0;
    while (newx = x) do
     begin
      newx := uniform(0.5*x,1.50*x);
      counter := counter+1;
       if (newx=x) and(counter=5) then
```

```
         begin
          if random_park <  0.5  then
           newx:=x-random_park
          else
           newx:=x+random_park
         end;
     end;
  end;
perturb_uni := newx;
 end;
 function perturb_eqi(x:integer) : integer;
 var
 newx : integer;
 counter : integer;
 begin
newx := x;
 if (x=0)  then
     begin
       if random_park <  0.5  then
         newx:=1
        else
         newx:=-1;
     end
 else
 begin
  counter:=0;
    while (newx = x) do
      begin
       newx := equilikely(trunc(x*0.5),trunc(1.50*x));
       counter := counter+1;
        if (newx=x)   and(counter=5)  then
           begin
            if random_park <  0.5  then
              newx:=x+1
             else
              newx:=x+1;
           end;
      end;
 end;
perturb_eqi := newx;
 end;
 function  test(val: integer): integer;
 const
   np = 11;
type
   gl array =  array [1..np]   of integer;
 var
   i    :integer;
   errcnt : integer;
   seed    : real;
   start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]   of integer;
    begin
  fmi := 1.0/fm;
  jstack := 0;
  l := 1;
  ir := n;
  fx := 0.0;
   while true do begin
   if (ir-l) < m then begin
(* < ============== 1*)
      for j := l+1 to ir do begin
 a := arr[j];
  for i := j-1 down to 1 do begin
      if (arr[i] < = a) then goto 11;
     arr[i+1] := arr[i];
  end;
 i := 0;
 11:    arr[i+1] := a
      end;
      if (jstack = 0) then goto 99;
     ir := istack[jstack];
     l := istack[jstack-1];
     jstack := jstack-2;
   end
   else begin
     i := l;
     j := ir;
     fx := (fx*fa+fc)/fm;
(* < ============== *)
        iq := l+(ir-l+1)* trunc(fx*fmi);
     a := arr[iq];
     arr[iq] := arr[l];
 21: if (j > 0) then begin
         if (a < arr[j]) then begin
   j := j-1;
    goto 21;
         end
      end;
      if (j < = i) then begin
        arr[i] := a;
(* < ============== *)
         goto 30;
      end;
     arr[i] := arr[j];
     i := i + 1;
 22:     if (i < = n) then
         if (a > arr[i]) then begin
      i := i + 1;
       goto 22;
  end;
      if (j < = i) then begin
        arr[j] := a;
        i := j;
         goto 30;
      end;
     arr[j] := arr[i];
     j := j - l;
(* < ============== *)
      goto 21;
  30: jstack := jstack + 2;
       if (jstack > nstack) then begin
  writeln('Overflow');
        end;
        if (ir -i) > = (i-l)  then  begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
  ir := i -1;
 (* < ========= *)
        end
        else  begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
        end
     end
    end;
99: end; (*  procedure process *)
 function anonymous(val : integer): integer;
 var
  arr   : gl array;
  n     : integer;
  i     : integer;
  count :  array[0..7]   of integer;
  correct :  array[0..11]   of integer;
  pos,k   : integer;
  error : boolean;
 begin (* anonymous *)
   i:= 1;
     while val < >  0  do  begin
      arr[i] := val   mod 8;
      val := val   div 8;
      i := i + 1;
    end;
   n := i -1;
     while i < = np  do  begin
      arr[i] := 0;
      i := i + 1;
    end;
   process(n,arr);
  val := 0;
   for i := np down to 1  do  begin
     val := val * 8 + arr[i]
   end;
  anonymous := val;
 end;  (* anonymous *)
 begin
   test := anonymous(val);
 end;


 function  test1(val: integer): integer;
 const
  np = 11;
type
  gl array =  array [1..np]   of integer;
 var
  i    :integer;
  errcnt : integer;
  seed   : real;
  start, finish : integer;
   procedure process(n: integer;  var arr: gl array);
    label
     11, 21, 22, 30, 99;
   const
```

```
     m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
   var
    l, jstack, j, ir, iq, i: integer;
    fx, fmi : real; a: integer;
    istack:  array[1..nstack]  of integer;
   begin
    fmi := 1.0/fm;
    jstack := 0;
    l := 1;
    ir := n;
    fx := 0.0;
    while true  do  begin
    if (ir-l) <  m  then  begin
yes1:=true;
         if perturb[1]  then
            begin
             ir := (perturb_eqi(ir));   (* < ===========1*)
             perturb[1] := false;
            end;
       for j := l+1  to ir  do  begin
  a := arr[j];
   for i := j-1 down to 1  do  begin
       if (arr[i] < = a)  then  goto 11;
      arr[i+1] := arr[i];
   end;
  i := 0;
  11:    arr[i+1] := a
       end;
       if (jstack = 0)  then  goto 99;
      ir := istack[jstack];
      l := istack[jstack-1];
      jstack := jstack-2;
    end
    else  begin
      i := l;
      j := ir;
      fx := (fx*fa+fc)/fm;
   (* < ===========2*)
         iq := l+(ir-l+1)* trunc(fx*fmi);
      a := arr[iq];


      arr[iq] := arr[l];
  21:  if (j > 0)  then  begin
          if (a <  arr[j])  then  begin
    j := j-1;
     goto 21;
          end
       end;
       if (j < = i)  then  begin
         arr[i] := a;
 (* < ===========  *)
          goto 30;
       end;
      arr[i] := arr[j];
      i := i + 1;
  22:     if (i < = n)  then
          if (a >  arr[i])  then  begin
       i := i + 1;
        goto 22;
   end;
```

```
       if (j < = i)  then  begin
         arr[j] := a;
         i := j;
          goto 30;
       end;
      arr[j] := arr[i];
      j := j - l;
 (* < ============ *)
       goto 21;
  30: jstack := jstack + 2;
       if (jstack >  nstack)  then  begin
  writeln('Overflow');
        end;
        if (ir -i) > = (i-l)  then  begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
  ir := i -1;
 (* < ============ *)
        end
        else  begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
        end
     end
    end;
99: end; (*  procedure process *)
 function anonymous(val : integer): integer;
 var
   arr   : gl array;
   n     : integer;
   i     : integer;
   count :  array[0..7]   of integer;
   correct :  array[0..11]   cf integer;
   pos,k   : integer;
   error : boolean;
 begin (* anonymous *)
    i:= 1;
      while val < >  0  do  begin
       arr[i] := val   mod 8;
       val := val   div 8;
       i := i + 1;
     end;
    n := i -1;
      while i < = np  do  begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
   val := 0;
    for i := np down to 1  do  begin
      val := val * 8 + arr[i]
    end;
   anonymous := val;
 end;  (* anonymous *)
 begin
    test1 := anonymous(val);
 end;
 function  test2(val: integer): integer;
 const
   np = 11;
```

```
type
 gl array = array [1..np]  of integer;
 var
   i   :integer;
   errcnt : integer;
   seed  : real;
   start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]  of integer;
    begin
     fmi := 1.0/fm;
     jstack := 0;
     l := 1;
     ir := n;
     fx := 0.0;
     while true  do  begin
     if (ir-l) <  m  then  begin
  (* < ========== *)
        for j := l+1  to ir  do  begin
   a := arr[j];
    for i := j-1 down to 1  do  begin
        if (arr[i] < = a)  then  goto 11;
       arr[i+1] := arr[i];
    end;
   i := 0;
   11:    arr[i+1] := a
        end;
        if (jstack = 0)  then  goto 99;
       ir := istack[jstack];
       l := istack[jstack-1];
       jstack := jstack-2;
     end
     else  begin
       i := l;
       j := ir;
       fx := (fx*fa+fc)/fm;
         if perturb[2]  then  begin
fx := perturb_uni(fx);   (* < ==========2 *)
        perturb[2] := false;
         end;
        yes2:=true;
          iq := l+(ir-l+1)* trunc(fx*fmi);
       a := arr[iq];
       arr[iq] := arr[l];
   21:  if (j > 0)  then  begin
          if (a <  arr[j])  then  begin
   j := j-1;
      goto 21;
           end
        end;
        if (j < = i)  then  begin
          arr[i] := a;
   (* < ========== *)
           goto 30;
       end;
      arr[i] := arr[j];
      i := i + 1;
  22:     if (i < = n) then
          if (a > arr[i]) then begin
       i := i + 1;
        goto 22;
   end;
       if (j < = i) then begin
         arr[j] := a;
         i := j;
          goto 30;
       end;
      arr[j] := arr[i];
      j := j - l;
 (* < ========== *)
       goto 21;
  30: jstack := jstack + 2;
       if (jstack > nstack) then begin
  writeln('Overflow');
        end;
        if (ir -i) > = (i-l) then begin
  istack[jstack] := ir;
  istack[jstack-1] := i + 1;
  ir := i -1;
 (* < ========== *)
        end
        else begin
  istack[jstack] := i-1;
  istack[jstack-1] := l;
  l := i+1;
        end
     end
    end;
99: end; (* procedure process *)
 function anonymous(val : integer): integer;
 var
   arr   : gl array;
   n     : integer;
   i     : integer;
   count :  array[0..7]  of integer;
   correct :  array[0..11]  of integer;
   pos,k  : integer;
   error : boolean;
 begin (* anonymous *)
    i:= 1;
      while val < > 0 do begin
       arr[i] := val  mod 8;
       val := val  div 8;
       i := i + 1;
     end;
    n := i -1;
      while i < = np do begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
   val := 0;
    for i := np down to 1 do begin
      val := val * 8 + arr[i]
    end;
  anonymous := val;
 end;  (* anonymous *)
 begin
   test2 := anonymous(val);
 end;


 function  test3(val: integer): integer;
 const
   np = 11;
type
   gl array =  array [1..np]   of integer;
 var
   i    :integer;
   errcnt : integer;
   seed   : real;
   start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]   of integer;
    begin
     fmi := 1.0/fm;
     jstack := 0;
     l := 1;
     ir := n;
     fx := 0.0;
     while true  do  begin
     if (ir-l) <  m  then  begin
(* < ============ *)
        for j := l+1  to ir  do  begin
   a := arr[j];
    for i := j-1 down to 1  do  begin
        if (arr[i] < = a)  then  goto 11;
      arr[i+1] := arr[i];
    end;
   i := 0;
   11:    arr[i+1] := a
        end;
        if (jstack = 0)  then  goto 99;
      ir := istack[jstack];
      l := istack[jstack-1];
      jstack := jstack-2;
     end
     else  begin
      i := l;
      j := ir;
      fx := (fx*fa+fc)/fm;
(* < ============ *)
         iq := l+(ir-l+1)* trunc(fx*fmi);
      a := arr[iq];
      arr[iq] := arr[l];
   21:  if (j > 0)  then  begin
         if (a <  arr[j])  then  begin
     j := j-1;
      goto 21;
         end
        end;
        if (j < = i)  then  begin
          arr[i] := a;
             yes3:=true;
              if perturb[3]  then  begin
             arr[i] := (perturb_eqi(arr[i]));   (* < ============ 3*)
             perturb[3] := false;
              end;
           goto 30;
        end;
       arr[i] := arr[j];
       i := i + 1;
   22:      if (i < = n)  then
           if (a >  arr[i])  then  begin
        i := i + 1;
         goto 22;
    end;
        if (j < = i)  then  begin
          arr[j] := a;
          i := j;
           goto 30;
        end;
       arr[j] := arr[i];
       j := j - l;
  (* < ========== *)
        goto 21;
   30: jstack := jstack + 2;
        if (jstack >  nstack)  then  begin
   writeln('Overflow');
         end;
         if (ir -i) > = (i-l)  then  begin
   istack[jstack] := ir;
   istack[jstack-1] := i + 1;
   ir := i -1;
  (* < ========== *)
         end
         else  begin
   istack[jstack] := i-1;
   istack[jstack-1] := l;
   l := i+1;
         end
       end
      end;
99: end; (*  procedure process *)
 function anonymous(val : integer): integer;
 var
   arr   : gl array;
   n     : integer;
   i     : integer;
   count :  array[0..7]   of integer;
   correct :  array[0..11]   of integer;
   pos,k   : integer;
   error : boolean;
 begin (* anonymous *)
     i:= 1;
       while val < >  0  do  begin
        arr[i] := val   mod 8;
        val := val   div 8;
        i := i + 1;
      end;
     n := i -1;
```

```
      while i < = np  do  begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
   val := 0;
    for i := np down to 1  do  begin
      val := val * 8 + arr[i]
    end;
   anonymous := val;
 end;  (* anonymous *)
 begin
    test3 := anonymous(val);
 end;
 function  test4(val: integer): integer;
 const
   np = 11;
type
   gl array =  array [1..np]   of integer;
 var
   i    :integer;
   errcnt : integer;
   seed   : real;
   start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]   of integer;
    begin
     fmi := 1.0/fm;
     jstack := 0;
     l := 1;
     ir := n;
     fx := 0.0;
     while true  do  begin
     if (ir-l) <  m  then  begin
  (* < ============ *)
        for j := l+1  to ir  do  begin
   a := arr[j];
    for i := j-1 down to 1  do  begin
        if (arr[i] < = a)  then  goto 11;
       arr[i+1] := arr[i];
    end;
   i := 0;
   11:    arr[i+1] := a
        end;
        if (jstack = 0)  then  goto 99;
       ir := istack[jstack];
       l := istack[jstack-1];
       jstack := jstack-2;
     end
     else  begin
       i := l;
       j := ir;
       fx := (fx*fa+fc)/fm;
  (* < ============ *)
```

```
      iq := l+(ir-l+1)* trunc(fx*fmi);
    a := arr[iq];
    arr[iq] := arr[l];
 21:  if (j >  0)  then  begin
       if (a <  arr[j])  then  begin
  j := j-1;
   goto 21;
       end
     end;
     if (j < = i)  then  begin
      arr[i] := a;
 (* < ============ *)
       goto 30;
     end;
    arr[i] := arr[j];
    i := i + 1;
 22:     if (i < = n)  then
       if (a >  arr[i])  then  begin
    i := i + 1;
     goto 22;
  end;
     if (j < = i)  then  begin
      arr[j] := a;
      i := j;
       goto 30;
     end;
    arr[j] := arr[i];
    j := j - l;
     yes4:=true;
if perturb[4]  then  begin
      j := (perturb_eqi(j));   (* < ============= 4*)
      perturb[4] := false;
       end;
     goto 21;
 30: jstack := jstack + 2;
     if (jstack >  nstack)  then  begin
 writeln('Overflow');
     end;
     if (ir -i) > = (i-l)  then  begin
 istack[jstack] := ir;
 istack[jstack-1] := i + 1;
 ir := i -1;
 (* < ============ *)
     end
     else  begin
 istack[jstack] := i-1;
 istack[jstack-1] := l;
 l := i+1;
     end
   end
   end;
99: end; (*  procedure process *)
function anonymous(val : integer): integer;
var
 arr   : gl array;
 n     : integer;
 i     : integer;
 count :  array[0..7]   of integer;
 correct :  array[0..11]   of integer;
 pos,k   : integer;
 error : boolean;
 begin (* anonymous *)
   i:= 1;
      while val < > 0 do begin
       arr[i] := val  mod 8;
       val := val  div 8;
       i := i + 1;
     end;
   n := i -1;
      while i < = np  do  begin
       arr[i] := 0;
       i := i + 1;
     end;
    process(n,arr);
   val := 0;
    for i := np down to 1  do  begin
      val := val * 8 + arr[i]
    end;
   anonymous := val;
 end;  (* anonymous *)
 begin
    test4 := anonymous(val);
 end;
 function  test5(val: integer): integer;
 const
   np = 11;
type
   gl array =  array [1..np]   of integer;
 var
   i    :integer;
   errcnt : integer;
   seed   : real;
   start, finish : integer;
    procedure process(n: integer;  var arr: gl array);
     label
      11, 21, 22, 30, 99;
    const
      m = 6; nstack = 50; fm = 7875; fa = 211.0; fc = 1663.0;
    var
     l, jstack, j, ir, iq, i: integer;
     fx, fmi : real; a: integer;
     istack:  array[1..nstack]   of integer;
    begin
     fmi := 1.0/fm;
     jstack := 0;
     l := 1;
     ir := n;
     fx := 0.0;
     while true  do  begin
       if (ir-l) <  m  then  begin
  (* < ========== *)
        for j := l+1  to ir  do  begin
   a := arr[j];
    for i := j-1 down to 1  do  begin
        if (arr[i] < = a)  then  goto 11;
       arr[i+1] := arr[i];
    end;
   i := 0;
   11:     arr[i+1] := a
        end;
        if (jstack = 0)  then  goto 99;
       ir := istack[jstack];
    l := istack[jstack-1];
    jstack := jstack-2;
   end
   else  begin
    i := l;
    j := ir;
    fx := (fx*fa+fc)/fm;
(* < ============= *)
      iq := l+(ir-l+1)* trunc(fx*fmi);
    a := arr[iq];
    arr[iq] := arr[l];
 21:  if (j >  0)  then  begin
       if (a <  arr[j])  then  begin
  j := j-1;
   goto 21;
       end
     end;
     if (j < = i)  then  begin
      arr[i] := a;
(* < ============= *)
       goto 30;
     end;
    arr[i] := arr[j];
    i := i + 1;
 22:     if (i < = n)  then
       if (a >  arr[i])  then  begin
     i := i + 1;
      goto 22;
  end;
     if (j < = i)  then  begin
      arr[j] := a;
      i := j;
       goto 30;
     end;
    arr[j] := arr[i];
    j := j - l;
(* < ============= *)
     goto 21;
 30: jstack := jstack + 2;
     if (jstack >  nstack)  then  begin
 writeln('Overflow');
      end;
      if (ir -i) > = (i-l)  then  begin
 istack[jstack] := ir;
 istack[jstack-1] := i + 1;
 ir := i -1;
         yes5:=true;
          if perturb[5]  then  begin
   ir := (perturb_eqi(ir));  (* < ============= 5*)
         perturb[5] := false;
          end;
      end
      else  begin
 istack[jstack] := i-1;
 istack[jstack-1] := l;
 l := i+1;
      end
    end
   end;
99: end; (*  procedure process *)
 function anonymous(val : integer): integer;
```

```
var
  arr   : gl array;
  n     : integer;
  i     : integer;
  count :  array[0..7]   of integer;
  correct :  array[0..11]   of integer;
  pos,k   : integer;
  error : boolean;
begin (* anonymous *)
  i:= 1;
    while val < >  0  do  begin
      arr[i] := val   mod 8;
      val := val   div 8;
      i := i + 1;
    end;
   n := i -1;
    while i < = np  do  begin
      arr[i] := 0;
      i := i + 1;
    end;
   process(n,arr);
  val := 0;
   for i := np down to 1  do  begin
     val := val * 8 + arr[i]
   end;
  anonymous := val;
end;  (* anonymous *)
begin
   test5 := anonymous(val);
end;
begin(************main**********************************)
seed := 123456722.0;
counter1:=0.0;
counter2:=0.0;
counter3:=0.0;
counter4:=0.0;
counter5:=0.0;
here1 := 0.0;
here2 := 0.0;
here3 := 0.0;
here4 := 0.0;
here5 := 0.0;
 for k := 1  to 250000 do
    begin
     inputvalue := equilikely(0,2400000);
     yes1 := false;
     yes2 := false;
     yes3 := false;
     yes4 := false;
     yes5 := false;
      for i := 1  to 5 do
        perturb[i] := true;
     sol :=  test(inputvalue);
     sol1 := test1(inputvalue);
     sol2 := test2(inputvalue);
     sol3 := test3(inputvalue);
     sol4 := test4(inputvalue);
     sol5 := test5(inputvalue);
      if yes1  then here1 := here1 + 1;
      if yes2  then here2 := here2 + 1;
      if yes3  then here3 := here3 + 1;
     if yes4  then here4 := here4 + 1;
     if yes5  then here5 := here5 + 1;
     if sol < >  sol1  then
       counter1 := counter1 + 1.0;
     if sol < >  sol2  then
       counter2 := counter2 + 1.0;
     if sol < >  sol3  then
       counter3 := counter3 + 1.0;
     if sol < >  sol4  then
       counter4 := counter4 + 1.0;
     if sol < >  sol5  then
       counter5 := counter5 + 1.0;
   end;
writeln('1 ', counter1 / here1:14:12);
writeln('2 ', counter2 / here2:14:12);
writeln('3 ', counter3 / here3:14:12);
writeln('4 ', counter4 / here4:14:12);
writeln('5 ', counter5 / here5:14:12);
 end.
```

# Appendix C

# Program Used in Experiment I

This code is from [2]. Function poidev is found on page 717, Function gammln is found on page 704, and the random number generator is from [1].

```
 procedure trial(xm : real);

 const
  pi = 3.141592654;

 var
 aa,bb,cc,dd,ee,ff,gloldm,glsq,glalxm,glg,em,t,y : real;
 seed : real;

 function random'park : real;
steve park's random number generator
 const
  a = 16807.0;
  m = 2147483647.0;
  q = 127773.0;
  r = 2836;
 var
   lo,hi,test : real;
 var
  hi := trunc(seed / q);
  lo := seed - q * hi;
  test := a * lo - r * hi;
   if test >   0  then
     seed := test
   else
     seed := test + m;
  random'park := seed / m;
 end;


function gammln(xx : real): real;
 const
   stp = 2.50662827465;
   half = 0.5;
   one = 1.0;
  fpf = 5.5;
var
x,tmp,ser : real;
j : integer;
cof :  array[1..6] of real;
var
cof[1] := 76.18009173;
cof[2] := -86.50532033;
cof[3] := 24.01409822;
cof[4] := -1.231739516;
cof[5] := 0.120858003e-2;
cof[6] := -0.53682e-5;
x:=xx-one;
tmp := x+fpf;
tmp := (x+half)*ln(tmp)-tmp;
ser := one;
 for j:= 1 to 6  do
   var
    x := x+one;
    ser := ser+cof[j]/x;
   end;
gammln := tmp+ln(stp*ser);
 end;
 var   (*main*)
  seed := 1234567.0;
  gloldm:=-1.0;
   if (xm <  12.0)  then
      var
        if (xm <> gloldm)  then
          var
           gloldm := xm;
           glg := exp(-xm);
          end;
        em := -1;
        t := 1.0;
         repeat
           em := em + 1.0;
           t := t * random˙park;
         until (t <= glg);
      end
      else
      var
         if (xm <> gloldm)  then
            var
             gloldm := xm;
             glsq := sqrt(2.0*xm);
             glalxm := ln(xm);
             glg := xm*glalxm-gammln(xm+1.0)
            end;
          repeat
            repeat
             y := pi*random˙park;
             y := sin(y)/cos(y);
             em := glsq*y+xm;
            until (em> =0.0);
           em := trunc(em);
           aa := sqr(y+1.0); 1
           bb := aa + 1.0;   2
           cc := em + 1.0;   3
           dd := gammln(cc); 4
           ee := em*glalxm-dd-glg; 5
        ff := bb * exp(ee); 6
        t := 0.9 * ff; 7
       until(random˙park <= t)
     end;

writeln(em);              this is output
 end;
```

# References

[1] STEPHEN K. PARK AND KEITH W. MILLER. Random number generators: good ones are hard to find. *Communications of the ACM*, October 1988.

[2] W. H. PRESS AND S. A. TEUKOLSKY AND B. P. FLANNERY AND W. T. VETTERLING. *Numerical Recipes The Art of Scientific Computing.* Cambridge University Press, 1986.

# Appendix D

# Notation and Symbols

This appendix contains the symbols used throughout the thesis.

$P$ input program to PIA

$P'$ the simplified input program

$P''$ the simplified input program after having each location identified

$P_x$ the x$^{th}$ version of program $P$

$[P\ ]$ the function program $P$ computes

$pred(l, i, x)$ the location executed before location $l$ on the i$^{th}$ interation of location $l$ on input $x$; this is a function of the input and the particular iteration of $l$ for the input

$succ(l, i, x)$ the location executed after location $l$ on the i$^{th}$ iteration of location $l$ on input $x$; this is a function of the input and the particular iteration of $l$ for the input

$ds_{\mathcal{S}_l}$ the succeeding data space of location $l$

$ds_{\mathcal{P}_l}$ the preceding data space of location $l$

$\mathcal{F}_l$ the class of faults used for infection analysis at location $l$

$\mathcal{F}$ the class of faults used for infection analysis regardless of location, i.e., $\bigcup_{i=1}^{n} \mathcal{F}_i$ for $n$ locations

$\mathbf{F}_{x,a_i}$ the failure propagation estimate for location $x$ and the i$^{th}$ active variable

$\mathbf{F}_x$ $= \min_i\{\mathrm{F}_{x,a_i}\}$ which is the failure propagation estimate of location $x$

$\mathbf{V}_{l,i}$ the viral propagation rate between data spaces $ds_{\mathcal{P}_l}$ and $ds_{\mathcal{S}_l}$ on the i$^{th}$ iteration of location $l$

$lfr_x$ the latent failure rate of path or path equivalence class $x$

$I_{x,a_i}$ the infection estimate for location $x$ and the i$^{th}$ semantic alternative

$I_x$ $= \min_i\{I_{x,a_i}\}$ which is the infection estimate of location $x$

$\phi$ the input domain of the program

$\phi_i$ the input domain for path equivalence class $i$

$E_i$ the probability of executing some abstraction level $i$ location or group of locations

$\mathcal{D}_l(x)$ the error degree of location $l$ for data state $x$; $x$ may be either the succeeding or preceding data state

$T$ the set of all trips through the program

$TS$ the set of all trip sets through the program

$h_i$ the dispersion histogram for path equivalence class $i$

$N_i$ the number or points uniformly chosen from $\phi_i$ for a given $\alpha$

$\hat{N}_i$ the number of points chosen using the modified algorithm[1] from $\phi_i$ for a given $\alpha$

$\gamma$ is $N_i - \hat{N}_i$; this is the difference in the number of points required for a given $\alpha$ between the modified algorithm and uniform sampling

$PEC$ the set of all path equivalence classes through a program

$\mathsf{A}_l$ the set of active variables of data space $l$; $l$ is the data space immediately succeeding location $l$

$\Delta$ the alternative class representing both the class of faults used for infection analysis and the perturbation function parameters used for propagation analysis

$\mathfrak{B}_{l,i}(x)$ the data state encountered prior to executing location $l$ on the i$^{th}$ iteration of location $l$ from input $x$

[1] Detailed in Chapter 5.

$\mathcal{A}_{l,i}(x)$ the data state produced after executing location $l$ on the i$^{th}$ iteration of location $l$ from input $x$

$\mathcal{B}_l(x)$ $= \bigcup_{i=1}^{n} \mathcal{B}_{l,i}(x)$ which is the set of all data states before location $l$ on input $x$

$\mathcal{A}_l(x)$ $= \bigcup_{i=1}^{n} \mathcal{A}_{l,i}(x)$ which is the set of all data states after location $l$ on input $x$

$\Upsilon$ the testing complexity metric

$\alpha$ the desired confidence in the probable correctness model

$R_P$ the reliability estimate for program $P$

$\lambda_P$ the failure probability of program $P$; the units are failures per number of executions

$\zeta_l$ a measure of how easily a location can hide faults relative to $\Delta$; the greater $\zeta_l$, the greater a location's ability

**Pr[A ]** the probability of event A occurring

**T** true

**F** false

$\emptyset$ the empty set

$a \Longrightarrow b$ $a$ implies $b$

$\wedge$ logical and

$\vee$ logical or

$[f](x) \uparrow$ the function computed by the code represented by $f$ on input $x$ is defined

# Bibliography

A. A. ABDEL-GHALY. Evaluation of competing software reliability predictions. *IEEE Transactions on Software Engineering SE-12*, pp. 950–967, September, 1986.

P. ALLEN AND MICHAEL DYER AND HARLAN D. MILLS. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, SE-12(No. 1):pp. 3–11, January, 1986.

C. W. BEDFORD AND E. E. HAMMOND, JR. AND G. W. BEST AND J. R. LUX. *Fields and Functions, A Course in Precalculus Mathematics.* Macmillan Company, 1970.

ALFS BERZTISS AND MARK A. ARDIS. *Formal Verification of Programs SEI Curriculum Module SEI-CM-20-1.0.* Software Engineering Institute, Carnegie-Mellon University, 1988.

BOEING COMPUTER SERVICES COMPANY SPACE AND MILITARY APPLICATIONS DIVISION. *Integrated Verification and Testing System IVTS System Maintenance Manual Volume V - Data-Flow Analysis.* August, 1983.

SUSAN S. BRILLIANT. *Analysis of Faults in a Multi-Version Software Experiment.* Technical Report, University of Virginia, May, 1985.

SUSAN S. BRILLIANT AND JOHN C. KNIGHT. Testing software using multiple versions, a preliminary report.

BENARD CARRE. *Lecture notes on Program Validation, Part 1: Algebraic Foundations.* University of Southampton Department of Electronics and Computer Science, December, 1982.

BENARD CARRE. *Lecture notes on Program Validation, Part 2: Program Flow Analysis.* University of Southampton Department of Electronics and Computer Science, December, 1982.

ByoungJu Choi and Aditya P. Mathur and Brian Pattison. P$^{M}$OTHRA: Scheduling Mutants For Execution on a Hypercube. *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification.* December, 1989, pp. 58-65.

Daniel I. A. Cohen. *Introduction to Computer Theory.* John Wiley and Sons, Inc., 1986.

Martin D. Davis and Elaine J. Weyuker. *Computability Complexity and Languages.* Academic Press, 1983.

R.A. DeMillo and D.S. Guindi and W.M. McCraken and A.J. Offutt and K.N. King. An extended overview of the mothra software testing environment. *Proceedings of Second Workshop on Software Testing Verification and Analysis*, July, 1988.

Richard A. DeMillo and Richard J. Lipton and Frederick G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, pp. 34–41, April, 1978.

Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

M. Fagan". Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal 15*, 1976, pp. 182-211.

John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pp. 156–173, June, 1975.

R. G. Hamlet. Testing programs with finite sets of data. *The Computer Journal*, 20(3).

Richard G. Hamlet. Probable correctness theory. *Information Processing Letters*, pp. 17–25, April, 1987.

Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 206–215, July, 1988.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pp. 576–583, October, 1969.

Ellis Horowitz. *Fundamentals of Programming Languages Second Edition.* Computer Science Press, 1984.

Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Inc., 1978.

WILLAM E. HOWDEN. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, July, 1977.

WILLAM E. HOWDEN. Completeness criteria for testing elementary program functions. *Proceedings Fifth International Conference on Software Engineering*, pp. 235–243, 1981.

WILLIAM E. HOWDEN. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, pp. 208–215, September 1976.

WILLIAM E. HOWDEN. Applicability of Software Validation Techniques to Scientific Programs. *ACM Transactions on Programming Languages and Systems*, July, 1980, pp.307-320.

J. C. HUANG. An approach to testing. *ACM Computing Surveys*, pp. 113–128, September, 1975.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 729-1983.* IEEE, 1983.

MICHAEL KARR AND DAVID B. LOVEMAN III. Incorporation of units into programming languages. *Communication of the ACM*, May, 1978, vol. 21, pp.385-391.

BRIAN W. KERNIGHAN AND P. J. PLAUGER. *The Elements of Programming Style.* McGraw-Hill Book Company, 1974.

BODGAN KOREL. Dynamic program slicing. *Information Processing Letters*, October, 1988.

BODGAN KOREL. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14(9), September, 1988.

BODGAN KOREL. The program dependence graph in static program testing. *Information Processing Letters*, January, 1987.

LANUSZ W. LASKI AND BODGAN KOREL. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3), May, 1983.

AVERILL M. LAW AND W. DAVID KELTON. *Simulation Modeling and Analysis.* McGraw-Hill Book Company, 1982.

N.G. LEVESON AND P.R. HARVEY. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, Spetember, 1983, pp. 569-579.

HARRY R. LEWIS AND CHRISTOS H. PAPADIMITRIOU. *Elements of the Theory of Computation.* Prentice-Hall, Inc., 1981.

SEYMOUR LIPSCHUTZ. *Schaum's Outline Series Theory and Problems of Probability.* McGraw-Hill Book Company, 1965.

B. LITTLEWOOD. Theories of software reliability: how good are they and how can they be improved? *IEEE Transactions on Software Engineering SE-6*, pp. 489–500, September, 1980.

F. H. MARTIN. HAL/S - The avionics programming system for shuttle. *Proceedings AIAA Conference on Computers in Aerospace*, November, 1977, pp. 308-318.

WILLIAM MENDENHALL AND RICHARD L. SCHEAFFER AND DENNIS D. WACKERLY. *Mathematical Statistics with Applications.* Duxbury Press, 1981.

DOUGLAS R. MILLER. Exponential order statistic models of software reliability growth. *IEEE Transactions on Software Engineering*, SE-12(No. 1):pp. 12–24, January, 1986.

EDWARD MILLER AND WILLIAM E. HOWDEN. *Tutorial: Software Testing and Validation Techniques.* IEEE Computer Society Press, 1981.

K. MILLER AND L. MORELL AND R. NOONAN AND S. PARK AND D. NICOL AND B. MURRILL AND J. VOAS. Estimating the failure probability of software when testing reveals no errors. *IEEE Transactions on Software Engineering*, (to be submitted 1990).

H. D. MILLS. On the statistical validation of computer programs. *IBM Federal Systems Division, Report FSC-72-6015, Gaithersburg, MD.*

HARLAN D. MILLS. *Software Productivity.* Little, Brown, and Company, 1983.

H. MILLS AND V.BASILI AND J. GANNON AND R. HAMLET. *Principles of Computer Programming: A Mathematical Approach.* Allyn and Bacon, 1987.

L. J. MORELL. A model for code-based testing schemes. *Fifth Annual Pacific Northwest Software Quality Conference*, pp. 309–326, 1987.

L. J. MORELL. Theoretical insights into fault-based testing. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 45–62, July, 1988.

LARRY J. MORELL. *Unit Testing and Analysis SEI Curriculum Module SEI-CM-9-1.0.* Technical Report, Software Engineering Institute, Carnegie-Mellon University, October, 1987.

LARRY JOE MORELL. *A Theory of Error-based Testing.* Technical Report TR-1395, University of Maryland, Department of Computer Science, April, 1984.

LARRY J. MORELL AND RICHARD G. HAMLET. *Error Propagation and Elimination in Computer Programs.* Technical Report TR-1065, University of Maryland, Department of Computer Science, July, 1981.

LARRY J. MORELL AND JEFFREY M. VOAS. *Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability.* Technical Report WM-88-2, College of William and Mary, Department of Computer Science, September, 1988.

MUSA AND IANNINO AND OKUMOTO. *Software Reliability Measurement Prediction Application.* McGraw-Hill, 1987.

LEON J. OSTERWEIL AND LLOYD D. FOSDICK. Dave-a validation error detection and documentation system for Fortran programs. *Software Practice and Experience*, pp. 473–486, October-December, 1976.

STEVE PARK. *Lecture notes on simulation*, Unpublished. College of William and Mary, August, 1988.

STEPHEN K. PARK AND KEITH W. MILLER. Random number generators: good ones are hard to find. *Communications of the ACM*, October, 1988.

TERRENCE W. PRATT. *Programming Languages Design and Implementation Second Edition.* Prentice-Hall Inc., 1984.

WILLIAM H. PRESS AND SAUL A. TEUKOLSKY AND BRIAN P. FLANNERY AND WILLIAM T. VETTERLING. *Numerical Recipes The Art of Scientific Computing.* Cambridge University Press, 1986.

ROGER S. PRESSMAN. *Software Engineering A Practitioner's Approach.* McGraw-Hill Book Company, 1982.

ROBERT L. PROBERT. Optimal Insertion of Software Probes in Well-Delimited Programs. *IEEE Transactions on Software Engineering*, January, 1982.

PETER W. PROTZEL. Automatically generated acceptance test: a software reliability experiment. *Second Workshop on Software Testing, Validation, and Analysis*, pp. 196–203, July, 1988.

C. V. RAMAMOORTHY AND F. B. BASTANI. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering SE-8*, pp. 543–371, July, 1982.

JOHN RUSHBY. *Quality Measures and Assurance for AI Software.* Technical Report NASA Contractor Report 4187, SRI International, Meno Park, CA., 1988.

STEPHEN R. SCHACH. *Software Engineering.* Aksen Associates Incorporated Publishers, 1990.

F. P. SCHOLZ. Software reliability modeling and analysis. *IEEE Transactions on Software Engineering,* Volume XII:pp. 25–31, January, 1986.

EUGENE H. SPAFFORD. Crisis and aftermath. *Communications of the ACM,* 32(6), June, 1989.

L.G. VALIANT. A theory of the learnable. *Communications of the ACM,* 27(11): pp. 1134–1142, November, 1984.

JEFFREY M. VOAS AND LARRY J. MORELL. Fault sensitivity analysis of computer programs. December, 1989, Unpublished abstract presented in New Ideas Session of TAV3-SIGSOFT 89.

JEFFREY M. VOAS AND LARRY J. MORELL. *Fault Sensitivity Analysis(PIA) Applied to Computer Programs.* Technical Report WM-89-4, College of William and Mary, Department of Computer Science, December, 1989.

JEFFREY M. VOAS AND LARRY J. MORELL. Propagation and infection analysis(PIA) applied to debugging. *IEEE Southeastcon '90 Record,* April, 1990.

JEFFREY M. VOAS AND LARRY J. MORELL. *Propagation Characteristics through Versions of One Program.* Technical Report WM-89-3, College of William and Mary, Department of Computer Science, October, 1989.

ELAINE J. WEYUKER. An empirical study of the complexity of data flow testing. *Second Workshop on Software Testing, Validation, and Analysis,* pp. 188–195, July, 1988.

ELAINE J. WEYUKER AND THOMAS J. OSTRAND. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering,* pp. 236–246, May, 1980.

LEE J. WHITE AND EDWARD I. COHEN. A domain strategy fo computer program testing. *IEEE Transactions on Software Engineering,* SE-6:pp. 247–257, May, 1980.

STEPHEN WOLFRAM. *Mathematica A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, 1988.

STEVEN J. ZEIL. Testing for Perturbations of Program Statements. *IEEE Transactions on Software Engineering SE-9,* May, 1983.

# Jeffrey Mark Voas

Jeffrey Mark Voas was born on August 16, 1963, in Reno, Nevada. He graduated from Biloxi High School in Mississippi in 1980. He then spent two years at Mississippi Gulf Coast Junior College. Mr. Voas then transferred to Tulane University, where he received his Bachelor of Science in Engineering degree in Computer Engineering and Mathematics in the Spring of 1985. In the Fall of 1985, he started graduate study at the College of William and Mary, where in December, 1986, he received a Master of Science degree in Computer Science. From Fall, 1987, until Spring, 1990, Mr. Voas has been enrolled in the doctoral program at the College of William and Mary studying computer science. Besides his thesis, he has been involved in two additional research projects, one involving *n*-version programming, and the other in ultra reliable software.

Mr. Voas is single, with plans to continue the research of the thesis through acquisition of a National Research Council post-doctoral fellowship at NASA/Langley. Plans after that are undetermined.