Probabilistic Program Execution is a Viable Way to Find Domains from Software

by

Jasdeep Kaur Sangha

Masters of Computer Science, Kurukshetra, 2006

Bachelor of Computer Applications, Kurukshetra, 2003

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2012

©Jasdeep Kaur Sangha 2012

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Probabilistic Program Execution is a Viable Way to Find Domains from Software

Master of Science 2012

Jasdeep Kaur Sangha

Computer Science

Ryerson University

Abstract

Program domains are useful in many areas of software engineering including software reliability, testing, and program comprehension. Program paths provide understanding of program dynamic behaviour. In this work, we show that it is possible to extract domains and the paths they represent from software using program execution based algorithms.

This thesis looks at five different execution based algorithms for identifying the domains/paths. These algorithms work differently to generate the domains from a possibly infinite set of possible paths. Two of the algorithms utilize an operational profile that describes the probability distribution of possible inputs. This allows them to generate the most important paths first.

These program execution based algorithms were explored using some simple functions. The results showed that the Probabilistic Execution algorithm produces the domains in the strictly most significant order, limited only by the equality of the integration available. The Monte Carlo Execution algorithm provided almost the same accuracy but is somewhat simpler. Of the algorithms that do not utilize operational profiles, Random Execution worked the best.

Acknowledgements

Thanks to staff and faculty at Ryerson University for their insights and support throughout my graduate studies. Thanks to Dr. Dave Mason for his mentoring and guidance. A special thanks to my family for their unwavering support. And finally, thank you Bir Singh, Harcharn Kaur and Dya Singh for your love and patience.

Contents

Declaration . . . iii
Abstract . . . v
Acknowledgements . . . vii
List of Tables . . . xiii
List of Figures . . . xv

1 Introduction **1**
1.1 Domain Analysis Terms . . . 2
1.1.1 Program . . . 2
1.1.2 Input Domains . . . 2
1.1.3 Predicate . . . 3
1.1.4 Program Paths . . . 3
1.2 Path/domain discovery . . . 3
1.3 Structure of the Thesis . . . 4
1.3.1 This chapter . . . 4
1.3.2 Related Work . . . 4
1.3.3 Methodology and Implementation . . . 4
1.3.4 Evaluation . . . 5
1.3.5 Conclusions and Future Work . . . 5

2 Related Work **7**
2.1 Background . . . 7
2.2 Software Reliability Engineering . . . 8
2.2.1 Software Reliability Engineering Process . . . 9
2.2.2 An Approach related to Software Reliability . . . 9
2.3 Domain Testing Strategies . . . 10
2.3.1 White and Cohen Strategy . . . 10
2.3.2 Another Domain Testing Strategy . . . 12

2.3.3 Domain Testing Strategy by Jeng et. al. 12
2.4 Program Comprehension 13
2.4.1 PREfix(defect detection tool) 13
2.4.2 Symbolic Path Simulation Algorithm 14
2.4.3 Path Slicing 15
2.4.4 Selective Path Profiling 16
2.4.5 Experimental Program Analysis 18
2.4.6 Monte-Carlo Method for Probabilistic Program Analysis 19

3 Methodology **21**
3.1 Program Execution Based Algorithms 22
3.1.1 The Probabilistic Execution Algorithm 22
3.1.2 The Monte Carlo Execution Algorithm 23
3.1.3 The Breadth-first Execution Algorithm 23
3.1.4 The Depth-first Execution Algorithm 23
3.1.5 The Random Execution Algorithm 24
3.2 Probability Density Functions 24
3.2.1 Introduction to Probability and PDFs 24
3.2.2 PDFs and Operational Profiles 24
3.2.3 PDFs and Path Predicates 25
3.2.4 Implemented PDFs 25
3.3 Numerical Integral of PDFs 27
3.3.1 Monte Carlo Integral of PDFs 28
3.3.2 Recursive Stratified Sampling 29
3.4 Experimental Set-up 30
3.4.1 An Artificial Finite Program 30
3.4.2 An Artificial Infinite program 31
3.4.3 Sine 32
3.4.4 Arctan 32

4 Evaluation **33**
4.1 An Artificial Finite Program 33
4.1.1 Breadth First, Depth First and Random Execution 34
4.1.2 Probabilistic and Monte Carlo Execution 37
4.2 An Artificial Infinite Program 42
4.2.1 Breadth First, Depth First and Random Execution 42

4.2.2 Probabilistic and Monte Carlo Execution 45
4.3 Sine 48
4.3.1 Breadth First, Depth First and Random Execution 48
4.3.2 Probabilistic and Monte Carlo Execution 51
4.4 ArcTan 54

5 Conclusions and Future Work **57**
5.1 Conclusions 57
5.2 Contributions 58
5.3 Limitations and Future Work 58

Bibliography **62**

List of Tables

4.1 Expected domains and their integrals for the finite program . . . 33
4.2 BreadthFirstExec takes 1224 milliseconds . . . 34
4.3 DepthFirstExec takes 1114 milliseconds . . . 35
4.4 RandomExec takes 1125 milliseconds . . . 36
4.5 ProbabilisticExec with PDFNormal takes 13847 milliseconds . . . 37
4.6 MonteCarloExec with PDFNormal takes 1106 milliseconds . . . 38
4.7 ProbabilisticExec with PDFWeibull takes 19399 milliseconds . . . 39
4.8 MonteCarloExec with PDFWeibull takes 3284 milliseconds . . . 39
4.9 ProbabilisticExec with PDFHistogram takes 10070 milliseconds . . . 40
4.10 MonteCarloExec with PDFHistogram takes 773 milliseconds . . . 41
4.11 BreadthFirstExec takes 85 milliseconds . . . 42
4.12 DepthFirstExec takes 76 milliseconds . . . 43
4.13 RandomExec takes 84 milliseconds . . . 44
4.14 ProbabilisticExec with PDFUniform takes 1607 milliseconds . . . 45
4.15 MonteCarloExec with PDFUniform takes 401 milliseconds . . . 46
4.16 ProbabilisticExec with PDFNormal takes 107978 milliseconds . . . 47
4.17 MonteCarloExec with PDFNormal takes 1197 milliseconds . . . 47
4.18 BreadthFirstExec takes 137 milliseconds . . . 48
4.19 DepthFirstExec takes 740 milliseconds . . . 49
4.20 RandomExec takes 661 milliseconds . . . 50
4.21 ProbabilisticExec with PDFNormal takes 10919 milliseconds . . . 51
4.22 MonteCarloExec with PDFNormal takes 3006 milliseconds . . . 53
4.23 BreadthFirstExec takes 653 milliseconds . . . 54

List of Figures

1.1 Execution based algorithms 2

3.1 Execution based algorithms 21
3.2 Normal - A Probability Density Function (1) 26
3.3 Weibull - A Probability Density Function (2) 27
3.4 Monte Carlo Integration (3) 29

4.1 Plot showing the domains covered by Breadth First. 34
4.2 Plot showing the domains covered by Depth First. 35
4.3 Plot showing the domains covered by Random Execution. 36
4.4 Plot showing the domains covered by Probabilistic Execution. 37
4.5 Plot showing the domains covered by Monte Carlo Execution. 38
4.6 Graph of one dimensional histogram 40
4.7 Plot showing the domains covered by Breadth First. 42
4.8 Plot showing the domains covered by Depth First. 43
4.9 Plot showing the domains covered by Random Execution 44
4.10 Plot showing the domains covered by Probabilistic Execution 45
4.11 Plot showing the domains covered by Monte Carlo Execution 46

Chapter 1

Introduction

My thesis is that Probabilistic Program Execution is a viable way to find paths and domains from software.

Program paths are useful for several problems in Software Engineering that includes testing, reliability, and estimating of resource consumption. Paths offer insight into a program's dynamic behaviour and uncover patterns of path locality that can be exploited to increase program performance. A path represents a sequence of instructions executed for a particular input value, determined by a sequence of predicates representing the program conditionals and loops. Our work concentrates on path discovery and five different execution based algorithms (15):

- Probabilistic execution.
- Monte Carlo execution.
- Random execution.
- Breadth-first execution.
- Depth-first execution.

These execution based algorithms work differently to generate the most important path from a potentially infinite set of possible paths. Domains are generated by following the paths, so we will just use domains to represent both paths and domains for the rest of the thesis. Domain discovery provides a practical approach to capture many important aspects of program's dynamic behaviour.

Recursive Stratified Sampling (RSS) algorithm is implemented in our work. This algorithm is used by the execution based algorithms and is a refinement of the Monte Carlo method that performs numerical integration using random numbers. We are using already implemented

Program Execution algorithms (15). We ran experiments to test the accuracy of the algorithms and the time taken to generate the domains. Four real Probability Density Functions (PDFs) are implemented which are integrated to obtain the probability that the random variable takes a value in the given integral.

Figure 1.1 shows program execution algorithms producing domains by using a program and an operational profile as inputs.

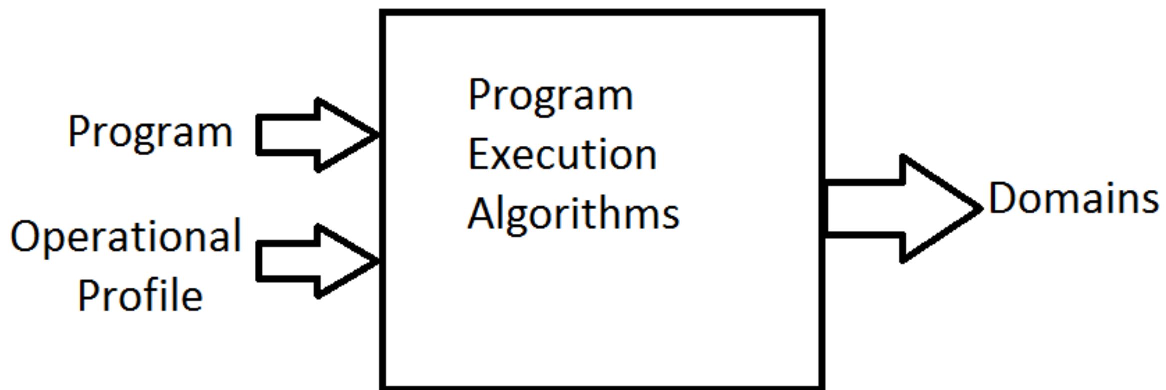

Figure 1.1: Execution based algorithms

1.1 Domain Analysis Terms

Domain Analysis provides a practical approach to capture many important aspects of program's dynamic behaviour. The terms related to domain analysis are described below.

1.1.1 Program

A program is a sequence of instructions written to perform a specified task. The two fundamental elements of a program relevant to this work are input domains and program paths.

1.1.2 Input Domains

The input domain of a program is the set of all possible input to that program. A domain can be represented by a set of predicates that represent the program conditionals and loops.

1.1.3 Predicate

A predicate is the conjugation of all of the conditional tests performed in the execution of a path. Every decision point of a computer program has an associated predicate which evaluates to true or false and its value determines which path is followed.

1.1.4 Program Paths

Program Paths (6) are distinct sequence of instructions that are executed from the start of the program to some point of interest based on a particular input value. They correspond to the flow of control in the program. Paths provide a source to reason about program's runtime behaviour. A Path is said to be feasible if there exists an input data that causes the program to execute the path properly. Otherwise, it is said to be infeasible.

For Example, consider the program fragment generating a path that adds even natural numbers from 1 to n:

```
procedure ex(int n)
  int sum = 0
  int j = 1
  while (j<=n)                        (P1)
    if ((j mod 2) = 0)                (P2)
     sum = sum + j                    (P3)
return sum                            (P4)
```

When invoked with an argument of 3, this function executes the path P1, P2, P1, P2, P3, P1, P2, P1, P4.

1.2 Path/domain discovery

The input domain of a program contains all possible sets of input data to that program. Path/domain discovery is important for various purposes in Software Engineering:

- Path Analysis is useful for testing purposes, but rarely seriously considered since the number of potential paths is too large and it becomes difficult to prioritize among those paths (15).
- Path/domain discovery is useful when estimating for software reliability

- When considering program comprehension, path discovery is useful to answer common questions about the code. Problems that developers face while understanding update paths are also handled by path analysis.

1.3 Structure of the Thesis

The structure of this section parallels the dissertation; each section corresponds to a chapter.

1.3.1 This chapter

1.3.2 Related Work

Program paths have been analysed using models for many different reasons including domain testing, reliability, maintenance and program comprehension. Recent work in many areas of computer science and engineering has shown that program paths provide a practical approach that offers insight into many aspects of a program's dynamic behavior.

Chapter 2 discusses the previous work related to path analysis that addresses various issues and describes the major approaches that handle those issues. The discussion includes strategies in detail, how they affect the issues and also some of the improvements in the methodologies.

1.3.3 Methodology and Implementation

In chapter 3, we look at various execution based algorithms used for domain discovery. We explain five different methods that have different ways of generating the domains from the potentially infinite set of possible domains.

All of the following algorithms generate sequences of program domains:

- Probabilistic Execution
- Monte Carlo Execution
- Random Execution
- Depth First Execution
- Breadth First Execution

The Recursive Stratified Sampling(RSS) algorithm is a form of Monte Carlo integration. RSS approximates the integral by recursively dividing the sub-regions until the error estimate

meets a specified tolerance. Four Probability density functions (PDFs) which we use as operational profiles (19), are described in detail. All the explored experiments are also described in this chapter.

1.3.4 Evaluation

Chapter 4 describes the experiments and the domains generated using the methods described in chapter 3. We also show the accuracy of the results and the time taken to produce them.

1.3.5 Conclusions and Future Work

Execution based algorithms are explored to find the domains from programs. Following are the contributions of this work:

1. a study of five execution based algorithms that work differently for the domain discovery.
2. implementation of RSS (Recursive Stratified Sampling) algorithm in smalltalk for approximation of the integral.
3. implementation of four probability density functions to use as an operational profile.
4. execution based algorithms were explored using some programs that test the accuracy of the algorithms and the time taken to generate the domains.

Chapter 5 discusses the conclusions derived from experiments. Limitations of the current approach and a possible avenue for exploration are also described in this chapter.

Chapter 2

Related Work

Domains/paths have been analysed in the past for several reasons that includes software reliability, testing, estimating of resource consumption and program comprehension. This chapter addresses the work that had been carried out by various authors concerning these categories.

2.1 Background

Dave Mason (16) provides an approach to reliability that makes it possible to compose component reliabilities in order to accurately determine system reliability. An important aspect of components is that system reliability can be estimated from the reliability of the components. First of all, sub-domains are looked upon that are provided by the program code. The most obvious domains would be the set of points that have the same stream of instructions executed upon them. The idea of identical instruction streams is captured in the definition of a path.

Operational Profile

An operational profile (19) simply consists of set of all operations that a system is designed to perform and the probabilities of their occurrence. Technically speaking, one can think of an operational profile as a quantitative characterization of how a system will be used. Operational profiles show the ways to increase the use, reliability, productivity and speed of development of a system.

Sub-domains

An input sub-domain is a subset of the input domain. Since the total input space of a system tends to be discontinuous, it should be partitioned into a series of domains that are each continuous. Dividing the input domain into subdomains is the concept of categorizing and grouping similar input values.

Path

Paths correspond to some flow of control in the program determined by the sequence of instructions executed from the start to some point of program to the successful termination or failure.

Dave Mason (16) mentioned an algorithm in his work to generate all paths in the shortest-first order. But according to him, an algorithm was needed that generates the most important paths first i.e. the paths with the highest likelihood of being executed. So a partially dynamic algorithm was generated (16). According to the author, "after having all the pieces for component reliability composition, accuracy of component reliability can be estimated". Probabilistic correctness for the component with the particular operational profile can be calculated after determining a set of program domains and their static properties by integrating over the program domain.

Program Domains

A Program domain is a subset of some specification domain and code domains and has a single computed result.

The author(16) believes that it is possible to calculate the reliability of a software system from the reliabilities of the components of which the system is composed but it is too time consuming to be practical.

2.2 Software Reliability Engineering

Software Reliability Engineering (18) is the standard and proven best practice that allows testers and developers to simultaneously:

- Ensure that product reliability meets user needs.
- Speed the product to market faster.
- Reduce product cost.
- Improve customer satisfaction and reduce the risk of angry users.
- Increase their productivity.

Software Reliability Engineering works by applying two fundamental ideas. Firstly, it delivers the desired functionality for the product under development much more efficiently. Secondly, it balances customer needs for reliability, development time, and cost, precisely and more effectively.

2.2.1 Software Reliability Engineering Process

The software reliability engineering process consists of five activities. They are:

- define ”necessary” reliability,
- develop operational profile,
- prepare for test,
- execute test, and
- apply failure data to guide decisions.

Software Reliability

Software Reliability determines the probability of failure in executing with a particular Operational Profile over a particular period of time.

Failure

A failure is the non-conformance of system behaviour with regards to user requirements and specifications. A software failure occurs during execution of a program by an incorrect or missing action.

2.2.2 An Approach related to Software Reliability

Input Domain Analysis for Software Reliability Measurement

(13) measures the reliability of a system based on the input sub-domains. The approach works by evaluating the specification and implementation of the program to determine the sub domains of the input.

The method that provides the appropriate reliability figures goes through the following steps:.

Generating Input Sub-Domains.

Generate functional sub-domains of the input which is achieved in two stages:

- The specification of the program is analysed to provide the initial set of sub-domains.
- The program is examined to break the code into sub-domains that are identified in stage 1.

After the completion of the two stages, an input sub domain has been associated to each transformation function.

Determine Reliability of sub-domains

The reliability of each of the identified sub-domains are measured. Then the reliability for the operational profile is also calculated.

Mason et.al. (13) conclude that order to calculate the reliabilities of system or component, it is necessary to subdivide the input space so the correctness can be checked within available sub-domains. So from this conclusion, we can say that reliability of a system is not possible without domain analysis, which is being done in our work.

2.3 Domain Testing Strategies

As mentioned earlier, domain analysis is also used for testing purposes. In a computer program, the input space is partitioned into a set of domains by control flow statements. Each of those domains corresponds to a program path which is executed by input data points. **Computer programs** can contain two types of errors which are:

- **Domain Error**: A Domain Error occurs when a specified input causes a wrong path to execute.
- **Computational Error**: Computational error occurs when the input data executes the right path, but the output value is wrong.

This section describes some domain testing strategies as part of our related work.

2.3.1 White and Cohen Strategy

The idea of Domain Testing was first studied by White and Cohen (23). Subsequently other authors proposed other strategies based on White and Cohen's strategy. The White and Cohen strategy is applied whenever a domain is subdivided into sub-domains by the some decision points in the program. Control flow statements in the program partition the input space into a set of related domains, each of which causes a path to execute. The purpose of domain testing methods are to select test data for program testing.

This domain testing strategy was initially proposed to detect domain errors but the by-product of the strategy is partial detection of computation errors. The boundary of each domain consists of border segments where each border segment is determined by a simple predicate. A border segment can be open or closed based on the relational operator in the predicate.

Assumptions of White and Cohen Strategy

- Path Predicates

  The important concepts of program path and path predicates together with domains were analysed, the result of which emphasizes on linear predicates instead of non linear, since in that situation, domains assume the simple form in the input space.

- Language considerations

  A number of programming language features can cause major complications in the detection of domain errors. The first feature is that of arrays that can cause complication. The second language feature is that of subroutines and functions which the authors excluded because of parameter passing and the side effects that can cause complications in the computation of control flow.

- Coincidental Correctness

  Coincidental Correctness occurs when a specific test point follows the wrong path but still gets the right output as if the test points are following the right path. White and Cohen assumed that this should not occur for the testing strategy.

Predicate Interpretations

In White and Cohen Strategy, every decision point of a computer program was accompanied by a predicate, whose value determined the path to be executed. The path condition was a compound condition that had to be satisfied by the input data to execute the control path. Not all the control paths that exist within the program were executable. If there was a path which is not followed by any of the input data, then the path was said to be infeasible and it was not followed in testing the program.

Predicate could be expressed both in terms of program variables and input variables. When a program variable available in the predicate was replaced by its symbolic value in terms of input variables, a predicate interpretation satisfying the original predicate was found.

Testing Strategy

The domain testing strategy did not emphasize the use of correct program to select the test points. The selected test points could be of two types:

- ON test point

  On test points lay on the boundary.

- OFF test point

This point is away from the boundary. In an open boundary, an OFF point was an interior point whereas in a closed boundary, OFF point was an exterior point.

It was shown that test points selected as above will give errors because of boundary shifts by causing an incorrect output. On the other hand, if the output was correct, then either the border was correct or if the border was incorrect, then the border could only belong to the line segments.

White and Cohen showed that their strategy effectively detects domain errors. Their strategy worked only on continuous spaces so that OFF point could be chosen close to the border. The strategy emphasized that number of test cases increased only linearly with the dimensionality of the input space. The strategy could be extended easily to n-dimensions.

From the above mentioned work, we can say that White and Cohen Strategy (23) checks any errors in the set of domains. The domains produced should have no errors so that a set of domains can cause the right path to execute. Our thesis concentrates on finding the most significant domains from programs to execute the right path.

2.3.2 Another Domain Testing Strategy

Clarke et.al.(8) proved that White and Cohen strategy was capable of detecting errors under certain well-defined conditions, but certain domain errors went undetected. So they proposed two strategies to improve their work by reducing the size of displaced areas. Two alternative strategies proposed by Clarke et. al. improved the error bound but with large number of inputs, the strategy may be impractical to use. It can only be applied to paths whose predicate interpretations are of low degree.

2.3.3 Domain Testing Strategy by Jeng et. al.

Jeng et.al.(10) extended the validity of the previous approaches by considering the restrictions in the earlier strategies. The authors argued that White and Cohen Strategy was extremely expensive since it requires large number of ON points resulting in large number of test cases to test the path. White and Cohen also argued that non linear predicates very rarely occur so they do not represent a significant problem. Their study was based on an informal survey of some programs. However, Jeng et.al. argued that non-linear predicates can occur by the result of linear predicates. They analysed some of the numerical programs and found that non linear predicate interpretations were common among the programs. Also it is common that some of the variables that are contained in the program need to be defined over discrete spaces.

The authors (10) examined the strategy from different perspective saying that detecting a domain error is same as determining if a border shift has occurred. Also, significantly fewer points are required in this methodology which significantly reduces the cost. In contrast to the previous strategies, Jeng et.al. strategy also requires constant number of test points which means test points are not dependent on the dimensions of the vertices on the given border. This strategy is applied to much wider class of applications since it also allows the spaces that are discrete and also not restricted to linear borders.

2.4 Program Comprehension

Program Comprehension is the process of understanding a program through documentation analysis(20). Although it is not uncommon for the programmers to work on unfamiliar code, understanding of the program enables the maintenance and successful evolution of the system. Many tools have been built in the past that achieve program comprehension through path analysis. Some of them are described below:

2.4.1 PREfix(defect detection tool)

Bush et.al.(7) presented an analyzer that detects programming errors because of the program's dynamic behavior. The analyzer follows the execution path of the program and provides valuable contexual information to the programmer, who needs to understand and repair the defects. According to the authors, PREfix is able to efficiently detect defects in large programs. They made the folowing observations:

- The majority of defects detected by PREfix were mismatched assumptions between multiple programmers working on same project.
- PREfix handled errors off main code paths.
- Their work found more errors than software engineering literature would expect for a particular size and type of application.
- Different coding styles gave different error rates.

Limitation

PREfix is a successful defect detection tool, which truncates the search space to achieve scalability. The simulation in the tool is conducted based on per-path and it does not support the join operation used by merge-based dataflow algorithm.

Although, PREfix is a successful tool, it does not support merge-based global analysis. Our next section will discuss about a path simulation algorithm which is more generic than PREfix.

From the above mentioned work, we can say that path analysis is really important which is being done in our thesis. In the above work (7), path analysis is done to detect programming errors because of program's dynamic behavior.

2.4.2 Symbolic Path Simulation Algorithm

Dor et.al.(9) presented an algorithm that handles dataflow analysis for tracking the flow of values through a program. The analysis is incorporated in ESP, a software validation tool for C/C++ programs that can be used to validate Windows operating system kernel against an important security property (Adams, Das, Yang). According to the authors, contributions of their work include the following:

- Value flow simulation, a new framework for inter-procedural, context-sensitive, path-sensitive value flow analysis.
- The notion of a value alias set is introduced that allows a client to perform strong updates even in the presence of memory aliasing.
- Value flow simulation is used in integration with ESP to validate the Windows operating system kernel against an important security property.

Value flow simulation is based on the "property simulation" algorithm and the difference between value flow simulation and property simulation algorithm that property simulation uses abstract finite state machine states as its dataflow facts, whereas value flow simulation uses value alias sets. "Value flow simulation algorithm dynamically creates and explores a space of analysis coordinates" (9). Their work was illustrated with an experiment in which value flow simulation was used in a client software validation tool, ESP.

ESP is a software validation tool for C/C++ programs that is run multiple times a week over tens of millions of lines of real, operating systems code. It uses value flow simulation to track one created value at a time. Many operating systems reserve a section of system memory for use by kernel mode components. The kernel provides a set of API functions that can be called from a user program, to perform various tasks. Some of these tasks require the kernel to read data from, or store data into, locations referenced by the pointer values passed in as arguments. "One possible source of security attacks on these systems is that a user program may pass in pointers that point into the reserved section, thereby duping the kernel into exposing or over-writing system data"(9). To prevent this form of attack, kernel code is written with the

guideline that all user mode pointers must be "probed", to ensure that they do not point into kernel memory, before they are dereferenced. ESP is used to ensure that there are no remaining dereferences of unprobed parameters in the kernel. The accuracy, efficiency and usability of ESP on the Probe property depends on the value flow analysis engine. The Windows kernel that is analyzed consists of roughly a million lines of code, spread over roughly 9000 functions. The kernel exposes an interface of several hundred callable functions, with roughly 500 pointer parameters. ESP reported errors on 30 pointers, and validated the rest. The experience of the author suggested that the analysis is both precise and scalable enough to serve as an engine for software validation tools.

2.4.3 Path Slicing

Jhala and Majumdar (11) presented a technique called path slicing which operates by evaluating subsets of the edges that are relevant towards demonstrating the reachability of the target location along a given path. As per the authors, "Path Slicing takes as input an infeasible path to a target location, and eliminates all the operations that are irrelevant towards the reachability of the target location"(11). PathSlice algorithm tracks the set of values at each point to evaluate if the suffix of the path from that point is viable and the source location of the last edge is added to the slice.

Limitation

Path slicing is limited because it lacks statical reasoning about termination. Because of this limitation, the viability of a path slice either guarantees reachability of target location or the program enters an infinite loop because of all the states that can execute the path slice.

Slicing is primarily studied in two forms:

- Static
- Dynamic

Path slicing is different from both static and dynamic slicing. It combines the precision of dynamic slicing with the static slicing's ability to reason about multiple paths. The algorithm is illustrated on a small imperative language with integer variables, references and functions with call-by-value parameter passing. The author begin by completely developing the technique for programs without procedure calls or references. Then pointers are added and then the path slicing is generalized to programs with procedure calls.

PathSlice algorithm has been implemented in the counterexample analysis phase of the software model checker BLAST. This algorithm was used to check file handling errors in a set

of application programs. The largest programs checked had about 100K lines of code. Without the path slicing algorithm, the counterexample analysis phase of BLAST did not scale to any of these examples, because the counterexamples were too large to analyze for feasibility, and second they contained many irrelevant reasons for infeasibility, causing a blowup in the size of the abstractions as well as the number of iterations taken to find the abstraction relevant to the property. Path slicing algorithm checked file handling errors in all the programs and found violations of the property in some of them. In general, slicing reduced counterexample traces to less than 1 percent of their original size in most cases. As the size of counterexample traces got bigger, the slicing becomes more effective. The end-to-end verification times for these examples were all below one hour. In the cases where tool returns a feasible path slice it is much easier for the user to go over the more succinct slice to ascertain the veracity of the counterexample. Thus, the experiments suggest that path slicing can extend the scope of static analysis by eliminating irrelevant details.

Above work (11) evaluates a path that is relevant in reaching towards the target location. Since the algorithm adds the source location of the viable point to the slice, the program can enter into an infinite loop when there are almost all the states that enters to the slice. This is the same situation that can occur with Breadth First Execution Algorithm in our thesis. Breadth first executes all the points to find the significant one and sometimes exhaustively searches the input space but could not find any significant domains.

2.4.4 Selective Path Profiling

Apiwattanapong and Harrold (5) presents an approach that examines subsets of a program entity for a powerful profiling technique- acyclic path profiling. The selective path profiling algorithm computes values for probes that guarantee that the sum of the assigned value along each acyclic path in the subset is unique, acyclic paths not in the subset may or may not have unique path sums. Before developing the algorithm, the use of several other approaches such as complete edge profiles and complete path profiles, for profiling acyclic paths selectively is examined. However, none of these approaches is effective for producing accurate profile of some selected paths.

Benefits

- The main benefit of SPP algorithm is that it facilitate monitoring of an important program entity-acyclic paths. Using SPP algorithm, the monitoring overhead, in terms of probes, consists of only those probes that are required to track the paths of interest. Because some large programs may have more than 232 acyclic paths profiling only those of interest

can provide significant savings.

- Another benefit of the approach is that the acyclic paths can be used to provide additional dynamic information.
- A third benefit of the approach is that it provides an opportunity to perform residual monitoring for path coverage effectively. The algorithm can be used to determine the required probes for the sets of uncovered paths.

The author adapted Ball and Larus (BL) algorithm to compute unique path sums for only paths of interest. The path-profiling algorithm developed by Ball and Larus, instruments a program so that, as the program executes, it records the number of times each acyclic path is executed. The algorithm consists of the following four steps, which are performed on each procedure that needs to be monitored:-

- creation of the representation,
- assignment of values to edges in the representation,
- selection of edges to be instrumented and computation of increments for instrumented edges and,
- optimization of the instrumentation.

Selective Path Profiling (SPP) algorithm uses the representation creates in Step one. The author implemented SPP and conducted two studies. For comparison, Ball and Larus algorithm (BL) is also implemented. The goal of study 1 is to measure the number of probes required by SPP for single paths in the directed acyclic graph (DAG) and compare it with the number of probes required by BL and find out if SPP could obtain savings in overhead. The goal of study 2 is to measure the number of probes required by SPP for a subset containing the small number of paths in the DAG, compare it with the number of probes required by BL and determine if SPP could obtain saving in overhead. The results of the studies show that there are significant savings using SPP algorithm when profiling a small subset of paths of interest; the results also show that the reduction is related to the structure of the procedure and thus, in some cases, SPP approach provides little savings.

The approach enables software developers to collect the required profile information while incurring only the necessary cost for the probes. The approach also provides an opportunity to profile systems that have limited memory resources limited disk space, or strict time constraints, and thus, cannot be fully instrumented. However, there are number of issues in optimizing selective path profiling that can be addressed. The author plans to extend the algorithm in

several ways. It can be extended to profile interprocedural paths to reduce more overhead when profiling them selectively.

2.4.5 Experimental Program Analysis

Ruthruff et. al. (22) reports the result of research that suggests that a new form of program analysis technique can be created by incorporating characteristics of experimentation into analysis. In this paradigm, descriptive and operational definitions of experimental program analysis, their illustration by example and several differences between experimental program analysis and experimentation in other fields is described. Authors show how the use of an experimental program analysis paradigm can help to identify limitations of analysis techniques, improve existing experimental program analysis technique and create new experimental program analysis techniques. Program analysis techniques support various software engineering activities such as testing, fault localization, impact analysis and program understanding. Authors argued that many program analysis approaches exist whose members are essentially experimental in nature. This means that these techniques can be characterized in terms of guidelines and methodologies defined and practiced within the paradigm of scientific experimentation. They showed how the experimental program analysis paradigm together with the operational definition, could help researchers identify limitations of analysis techniques, improve existing program analysis techniques and create new experimental program analysis techniques.

Applications

- Assessing and Improving EPA Techniques: Program analysis approach provides a way for assessing the limitations of the existing EPA techniques.

- Creating New EPA Techniques: The operational definition of experimental program analysis provides a means for creating new EPA techniques.

It is shown that by following experimental program analysis as a new program analysis paradigm, it is possible to identify limitations of EPA techniques, improve existing EPA techniques and create new EPA techniques.

This work(22) is also doing program analysis which we are doing in our thesis. Our thesis concentrates on execution based program analysis whereas this work(22) does experimental program analysis and focuses on improving the existing and creating the new Experimental Program Analysis (EPA) techniques.

2.4.6 Monte-Carlo Method for Probabilistic Program Analysis

Davis Monniaux (17) introduced a new method for the analysis of programs featuring both probabilistic and non-probabilistic non-determinism which is a combination of random testing and abstract interpretation. This method takes both randomness and ordinary non-determinism into account. This paper is encouraged by the fact that sometimes probability of certain outcomes of a randomized computation process need to be estimated. So it becomes particularly important to obtain upper bounds on the probability of failure. Non-determinism is incorporated to account for the behaviour of inputs that cannot be modelled by random distributions. According to the Abstract Monte-Carlo method, it seems that it is easy to use any abstract interpreter. However, some precautions need to be taken in the presence of calls to random generators. The author of this paper is proposing a semantics about deterministic programs taking one input x chosen according to some random distribution and one input y in some domain. But it is shown that such semantics is not very suitable for program analysis because it tracks the number of calls made to number generators.

Another semantics is proposed that identifies occurrences of random generators by their program location and loop indices. The analysis algorithm of this paper is a randomized version of an ordinary abstract interpreter. In fact, the calls to random generators are treated as follows:-

- calls occurring inside fix point convergence iterations are interpreted as constants chosen randomly by the interpreter;
- calls occurring inside fix point convergence iterations are interpreted as upper approximations of the whole domain of values the random generator yield.

Complexity The complexity of the method is the product of two independent factors:

- the complexity of one ordinary static analysis of the program: - this depends not only on the program but on the random choices made.
- the number of iterations: - this depends only on the requested confidence interval.

The generic method combines the well known techniques of abstract interpretation and Monte-Carlo program testing into an analysis scheme for probabilistic and non-determinism programs. This method is mathematically proven correct, and uses no assumption apart from the distributions and non-determinism domains supplied by the user. The method has been implemented on top of a simple static analyzer and the experiments showed interesting results on small programs.

This work (17) is also doing program analysis based on both probabilistic and non-probabilistic methods. Program analysis is done to estimate the probability of failure based on the probability of some outcomes of a randomized computation process.

Chapter 3

Methodology

In Chapter 2, we described the work related to path analysis which is used for many different purposes. In this thesis, domain discovery is done to extract the domains from the software using execution based algorithms. Two of the algorithms utilize an operational profile describing the probability distribution of possible inputs.

In addition to the necessary program and the operational profile used by some program execution algorithms, the algorithms have also been parameterized by the integrator used to evaluate the integrals of the operational profile. Figure 3.1 shows execution based algorithms producing domains.

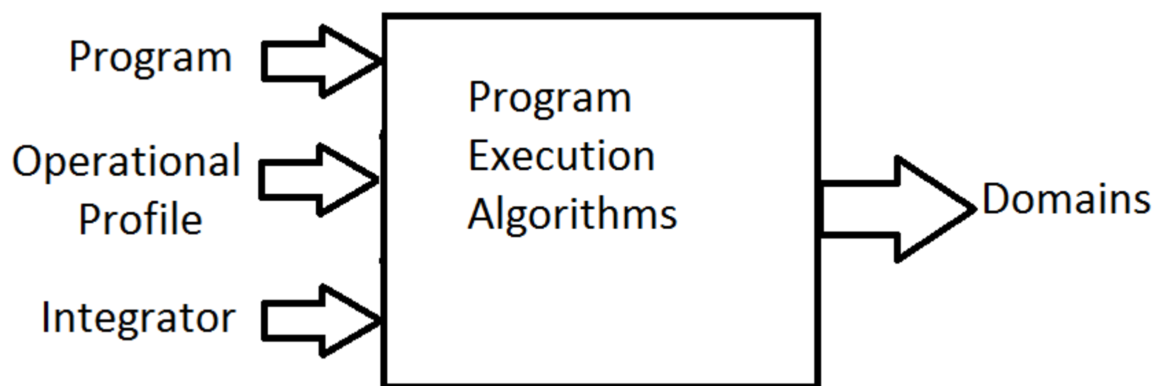

Figure 3.1: Execution based algorithms

3.1 Program Execution Based Algorithms

Our work looks at 5 different program execution algorithms for identifying the domains. Two algorithms require probabilistic input parameters which when combined with a set of other probabilistic predicates build a domain. The probabilistic input parameters only have probabilistic meaning rather than truth or falsity. So they have a particular probability of input values being in the input domain, based on the operational profile.

Probabilistic execution of a program looks at values differently. In normal program execution, all variables and expressions actually contain numeric values but in a probabilistic execution, variables can contain either numeric values or expressions of the (probabilistic) input parameters.

3.1.1 The Probabilistic Execution Algorithm

Probabilistic Execution is a continuation based algorithm that works by utilizing an operational profile and by capturing continuations. An operational profile describes the probability distribution of possible inputs. The first domain produced by the algorithm is exactly the most frequent domain. In addition, the frequency is available as soon as the domain is produced. Then the next generated domain is the next most frequent, and so on.

Operational Profile

An **Operation Profile** is a statistical description of the context in which the program will be used. It models the inputs of the system that will ideally be based on the actual field usage. One can think of it as a multi-variate probability density function which can be defined as a histogram or a continuous function. A PDF is non-negative at all points and the integral over the entire input space is 1. This is further defined in section 3.2.

Continuation

A **Continuation** represents the execution state at any given point. Capturing a continuation lets the program perform some other tasks and then return to the continuation to resume execution from the previous state. There can be many captured continuations pending future execution at any time.

The algorithm captures the continuation whenever a probabilistic predicate has to be evaluated to determine program flow. The algorithm maintains a priority queue based on frequency. When a probabilistic predicate needs to be evaluated to decide which paths to follow, a continuation is captured and two paths are placed on the queue (15). The highest frequency path on the priority queue is then popped and the current predicate is set to the popped path and execution is resumed.

3.1.2 The Monte Carlo Execution Algorithm

Monte Carlo Execution works by randomly generating a datapoint in the input space as determined by the operational profile to choose the value from the more-probable sub-domains. It then tests the points against already covered domains to date. If that domain has already been discovered, it generates another point. Otherwise, the program is executed with the generated point. Probabilistic predicates are generated following the same method as probabilistic execution. To determine which paths to follow, no continuations are used, instead list of current predicates is augmented by the new predicate and execution is resumed based on the truth of the predicate to let the execution go forward in the proper direction.

Both Probabilistic and Monte Carlo Algorithms have their own **advantages** and **disadvantages**. The major drawbacks of Probabilistic Execution are performance based: Firstly, it depends on capturing the continuation at the point where a probabilistic predicate needs to be evaluated to determine which paths to follow. Secondly, the exactness of the frequency is based on the accuracy of the integration of the PDF used as determined by the operational profile.

The Monte Carlo Algorithm is somewhat simpler and less expensive than the probabilistic algorithm since no integral is calculated and no continuations are captured. The domains generated by this algorithm is not strictly ordered so that first domain produced may not be strictly the most frequent domain and frequency is also not calculated after a domain is produced, although an integral can be calculated over the discovered domains.

3.1.3 The Breadth-first Execution Algorithm

Breadth first execution ignores the operational profile and works based on a breadth-first search of the program structure. The algorithm begins at the root node and captures one continuation and add it to the end of the queue and executes the path. When the current execution completes, it removes the first item from the queue if it is not empty. So the algorithm may run into infinite path lengths since it takes no account of the input values. It may end up with having many insignificant domains.

3.1.4 The Depth-first Execution Algorithm

Depth first execution also ignores the operational profile and expands the first input that appears and goes deeper and deeper into the problem space. This algorithm is also based on capturing the continuation and adds it to the beginning of the queue. When the current execution completes, it executes the first item from the queue if the queue is not empty. As for breadth first, this algorithm may also run almost immediately into potentially infinite loops and hence infinite path lengths.

3.1.5 The Random Execution Algorithm

Random Execution is a compromise or a combination of both Breadth First and Depth First since it is not in a specific order. It decides randomly at each stage as to which algorithm (breadth first or depth first) should be used. So it works by sometimes searching the neighbours and sometimes the deeper paths. So it doesn't get stuck in looking for low frequency paths, however it may go part way down improbable paths.

3.2 Probability Density Functions

3.2.1 Introduction to Probability and PDFs

To accurately identify domains in a program, we need to know the values that the variables and the expressions in a program can take on.

In this section, we will examine the concept of Probability Density Functions and how we use them to model variables and expressions in a programming language.

Probability Density Function

A Probability Density Function (14) of a continuous random variable is a function which can be integrated to obtain the probability that the random variable takes a value in the give integral. In other terms, the probability for the random variable to fall within a particular region is given by the integral of the variable's density over that region. A PDF is a non-negative function with an integral of 1. A function f is said to be a PDF iff:

$$f(x) \geq 0, \forall x \in domain(f), and, \int_{domain(f)} f(x)\,\mathrm{d}x = 1 \tag{3.1}$$

It is the probability that the random value x of the function f(x) is chosen and the integral over the entire space must be 1. PDFs can be either discrete or continuous but our work focuses only on continuous domains.

3.2.2 PDFs and Operational Profiles

A program under analysis will be used in the context of an operational profile which in our case is a PDF. It will describe the probability of a particular random variable to fall within a given region. The probability is given by the integral of the PDF's density over the input space.

3.2.3 PDFs and Path Predicates

The probability that a particular random variable falls within a given region is determined by weighting of an operational profile that covers the domain. A domain is represented by a predicate which is the conjugation of all of the conditional tests performed at some point in the evaluation of a path. Every decision point of a computer program has an association with a predicate which evaluates to true or false and its value determines which path is followed.

3.2.4 Implemented PDFs

Four Probability Density Functions are implemented to obtain the probability for the random variable to fall within a particular region.

Normal Distribution

Normal Distribution is a continuous probability distribution having a bell-shaped probability density function. A random variable x is said to be normally distributed with mean μ and variance σ^2 using the following formula(1):

$$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}} (\frac{x-\mu}{\sigma})^2 \tag{3.2}$$

where the parameter μ is the mean: the location of the peak and σ is known as the standard deviation. Normal Distribution is commonly used in probability literature (12). Figure 3.1 shows the graph of probability density function of normal distribution with different mean and variance values.

The red curve is the standard normal curve. Both mean and the variance are varied as shown in the graph.

Weibull Distribution

Weibull Distribution is also a continuous probability distribution that can take on the characteristics of other types of distributions, based on the value of the shape parameter. It is popular in the reliability literature. The probability density function of Weibull Distribution is defined as follows (2):

$$f(x;\lambda,k) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1} e^{-(x/\lambda)^k} \tag{3.3}$$

where k and λ are the shape and the scale parameters respectively. The values of both shape and scale parameters should be greater than 0.

Figure 3.2 is the graph of 2-parameter pdf of Weibull Distribution with different shape and scale values. $f(x;\lambda,k)$ is plotted vs x.

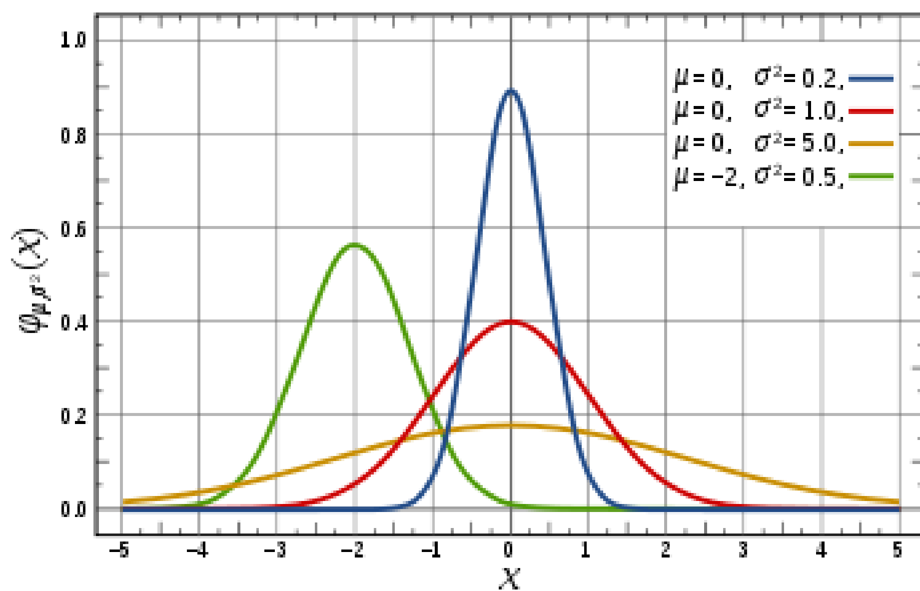

Figure 3.2: Normal - A Probability Density Function (1)

Uniform Distribution

Uniform Distribution is the third distribution where the random variables have the same probability of occurrences at all points in the input space. The uniform distribution is continuous over a range which can be denoted as: $[a, b]$

Histograms

Histograms are the fourth probability density function which is integrated to obtain the probability that a random variable takes a value in the given integral. Unlike the other three PDFs, histograms can be multi-dimensional PDFs. A histogram is a graphical way of presenting data which has been collected in categories.

In general, a histogram is a vertical bar chart that shows a “visual impression of the distribution of data” (4). It gives the “estimate of the probability distribution of a continuous random variable”. Histograms are used to represent large sets of data which can be organised and displayed in a user friendly format. The total volume of a histogram used for a PDF is 1.

PDF is a function that tells the relative likelihood for a random variable x to take on a given value. However, the probability for the random variable to fall within a particular region is given by the integral of the variable’s density over that region. In the next section, we are describing the integration methods used in our work.

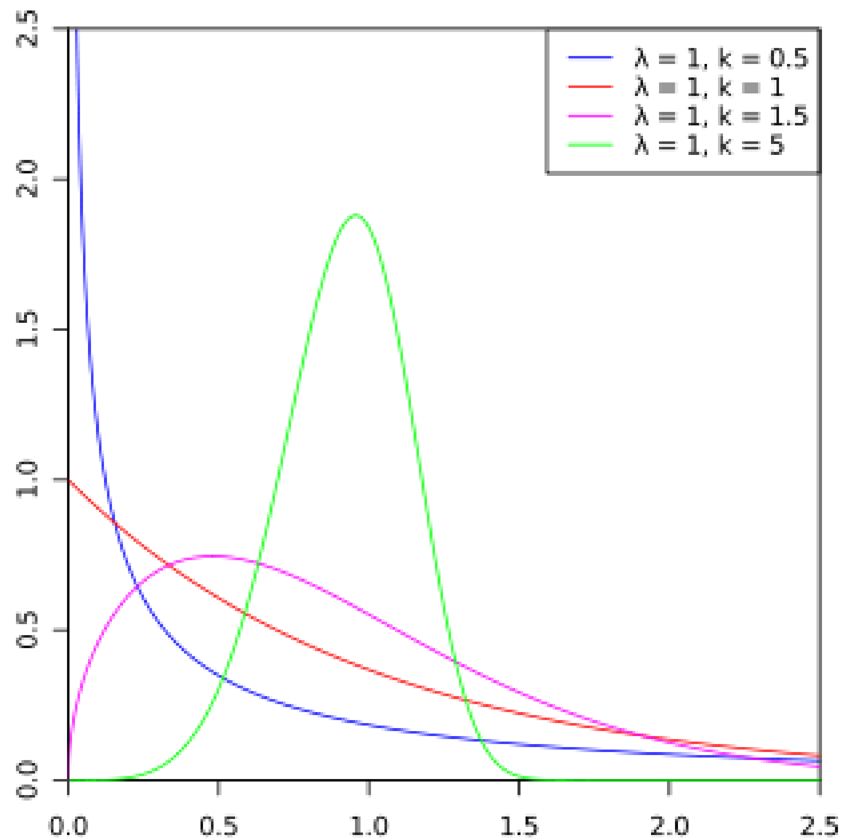

Figure 3.3: Weibull - A Probability Density Function (2)

3.3 Numerical Integral of PDFs

Numerical integration is a technique to calculate the integral using numerical techniques rather than analytical techniques. Numerical integration is also sometimes called quadrature. There are several algorithms to calculate the numerical value of a definite integral. The algorithms do the evaluation of the integrand at some set of points called integration points and the combination of those points give the integral. The error in the numerical integration method is determined by the number of evaluations of the integrand and the reducing the number of evaluations reduces a total round off error. We will describe a **Monte Carlo Integration** method for the evaluation of definite integrals, which can be used for any dimension.

3.3.1 Monte Carlo Integral of PDFs

Monte Carlo integration (3) is a method for "numerical integration using random numbers" (3) which is done by choosing a finite bounding box, the sample space, that encloses the target region for which the integration is required. The Plain Monte Carlo Integration algorithm uses the formula (3),

$$I = V * \left(\frac{1}{N}\right) \sum_{i=1}^{N} f(\bar{x_i}) = V\langle f \rangle \tag{3.4}$$

where,

$$V = \int_{v} \mathrm{d}\bar{x} \tag{3.5}$$

the volume of the integration region v

N = number of sample points and,

$\langle f \rangle$ = the mean of the sample points that are generated randomly using a random number generator.

The algorithm computes an estimate of the integral and the error of a multidimensional definite integral of the form,

$$\int_a^b \mathrm{d}x_1 \int_x^y \mathrm{d}x_2 ...... \int_p^q \mathrm{d}x_3 f(x_1, x_2, ...x_n) = \int_V f(\bar{x})\, \mathrm{d}\bar{x} \tag{3.6}$$

where

$$\bar{x} = x_1, ...x_n \tag{3.7}$$

and volume V is the region of integration.

The algorithm distributes the points uniformly over the integration region. The error estimate of the algorithm is not said to be "strict error bound" (3) which is a serious weakness since random sampling of the integration region "may not uncover all the important features of the function resulting in an underestimate of the error samples."(3) Two popular methods used to deal with this weakness are Recursive Stratified Sampling and Importance Sampling. In our work, we focus on **Recursive Stratified Sampling** which is an adaptation of traditional Monte Carlo algorithm.

Figure 3.3 illustrates the Monte Carlo Integration. The circle inside the figure is the function used and the points generated in the circle are all distributed uniformly over the entire region including the square. So the area of the circle is estimated by the "ratio of the points inside

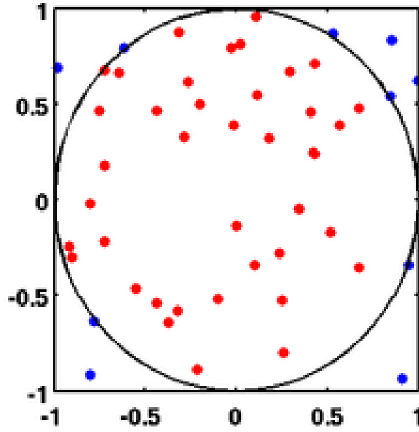

Figure 3.4: Monte Carlo Integration (3)

the circle to the total number of points" (3).

3.3.2 Recursive Stratified Sampling

The Recursive Stratified Sampling algorithm (21) is a form of Monte Carlo integration. The algorithm checks the error and if it is too large, "the region is subdivided and the procedure is recursively applied to sub-volumes" (3) until the error estimate meets a specified tolerance. The subdivisions are not applied to all bisections, instead the algorithm concentrates over the area with largest variance of the function.

Suppose the volume V is subdivided into two equal sub-volumes denoted by a and b and $N/2$ points are sampled in each sub-volume, then

$$\langle f\rangle' = \frac{1}{2}(\langle f\rangle_a + \langle f\rangle_b) \tag{3.8}$$

where

$$\langle f\rangle_i = \frac{1}{N}\sum_i f(x_i)_i \tag{3.9}$$

Then the variance (21) of the above is given as:

$$Var(\langle f\rangle') = \frac{1}{4}[\frac{Var_a(f)}{N_a} + \frac{Var_b(f)}{N - N_a}] \tag{3.10}$$

The algorithm evaluates $\langle f\rangle$ in the rectangular region $R = (x_a, x_b)$, which consists of two coordinate vectors of its diagonally opposite corners. The variances in each of the subregions

are estimated to find the most favorable dimension i to bisect for which the combined variance of two sub-regions is minimized. The variance in each of the sub-regions is estimated by allocating a fraction of total number of available points. The same procedure is followed recursively for each of the two sub-regions from the best bisection The integral and the error estimates when combined gives overall result.

3.4 Experimental Set-up

Our work concentrates on finding domains/paths from the program. In this section, we will describe the experiments that measure our results and the actual results will be reported in the next chapter. The experimental-programs that we are using to find the domains are:

1. a very simple program with finite number of domains
2. another artificial program with infinite number of domains
3. Sine
4. ArcTan

Each of the probabilistic program execution algorithms is run using these programs in order to show the different orders of the domains found. We will discuss all four programs used for the experiments in the next sections.

3.4.1 An Artificial Finite Program

This is an experimental program used to find the domains. For both Probabilistic and Monte Carlo Execution, the program uses PDF Normal, PDF Weibull and one dimensional Histogram as an operational profile. The sample code of this program is:

```
procedure finiteprogram(probVar x)
   int i = 0
   int j = 1
   while (j<=10)
     if((j-1 <= x) && (x<j))
        i=j
   return i
```

The parameter x in this code is a probabilistic variable which is compared each time the loop increments with the values $i := 1, 2, 3..10$. With this program, the algorithms will find the 11

domains. The domains will be starting from 0 to 1, 1 to 2, 2 to 3 and so on up to 10 to 11 since minimum amd maximum range we are using will be 0.0 and 11.0 respectively. We will show the orders of those domains found using each algorithm in the next chapter.

3.4.2 An Artificial Infinite program

This is also an experimental program used to find the domains. For both Probabilistic and Monte Carlo Execution, the program is used with both Normal and Uniform distribution. The sample code of this program is:

```
procedure infiniteProgram (probVar x)
   int j = 1
   int i = 0
   if(x>=5)
      if(j mod 2) = 0
         while(1000.0/(j+5))>x
            if(x > 7)
               i = j
            else
               i = -j
            j = j + 1.0
      else
         while(100.0/(j+5))>x
            if(x > 7)
               i = j
            else
               i = -j
            j = j + 1.0
  else
      if(x > 9)^(x < 6)
         return i
      else
         return i + 1
```

The probabilistic variable in this example is also compared each time the loop is incremented with the values $j := 1, 2, 3...10$. The domains we expect to find will be in the range from 1 to 11 since the total input space ranges from 1 to 11. The orders of the domains found using each algorithm are shown in detail in the next chapter.

3.4.3 Sine

The trigonometric function, Sine, is used as a program to run the experiments. Domains are found from sine using Normal Distribution as a pdf. The actual block that calls the Sine function is:

```
[:x | (PPEDemoFloat new value: x) sin]
```

The above code calls the sin method which is in the class PPEDemoFloat. The sin method in PPEDemoFloat class is copied from built-in sin method. We are not directly calling built-in sin since it is defined as a primitive. So instead of executing the method, the primitive would be executed and our algorithms can only capture the domains of smalltalk code, not primitives.

For the algorithms that use operational profile, Sine is run Normal distribution whose $\mu = \frac{\pi}{4}$ and $\sigma = 15$.

3.4.4 Arctan

Arctan is a two-dimensional function which is also used as a program to run the experiments. Domains are found from Arctan using histogram as a probability density function. The code that calls ArcTan function is:

```
[:x :y| (PPEDemoFloat new value: x) arcTan: y]
```

The above code calls the arctan method which is in PPEDemoFloat class. The reason for not directly executing the built-in arctan is the same as mentioned in the previous section.

Both sine and arctan are used with each of the program execution algorithms to find the domains. Normal distribution, Weibull distribution and histograms are used as probability density functions for both sine and arctan programs. The performance of the algorithms is examined based on the run time and accuracy of the domains found, which we will explore in the next chapter.

Chapter 4

Evaluation

In this chapter, we will report the results of our work with reference to the methodology discussed in the previous chapter. Our work focuses on Program Execution Algorithms that finds the domains from the software. All the algorithms are run using four programs which are described in the previous chapter.

4.1 An Artificial Finite Program

An artificial example is run using each algorithm and the order of the domains found are shown in the tables in section 4.1.1. Table 4.1 shows the domains expected from this finite program and their integrals. The integral for the domains are calculated using the normal distribution

Number	Domains	Integral
1	$0 \leq x \wedge 1.00000 > x$	0.00924
2	$1.00000 \leq x \wedge 2.00000 > x$	0.02783
3	$2.00000 \leq x \wedge 3.00000 > x$	0.06559
4	$3.00000 \leq x \wedge 4.00000 > x$	0.12097
5	$4.00000 \leq x \wedge 5.00000 > x$	0.17466
6	$5.00000 \leq x \wedge 6.00000 > x$	0.19741
7	$6.00000 \leq x \wedge 7.00000 > x$	0.17466
8	$7.00000 \leq x \wedge 8.00000 > x$	0.12097
9	$8.00000 \leq x \wedge 9.00000 > x$	0.06559
10	$9.00000 \leq x \wedge 10.00000 > x$	0.02783
11	$10.00000 \leq x \wedge 11.00000 \geq x$	0.00924

Table 4.1: Expected domains and their integrals for the finite program

table. The values of μ and σ are the same ($\mu = 5.5$ and $\sigma = 2$) as used to run the Probabilistic algorithms with this program.

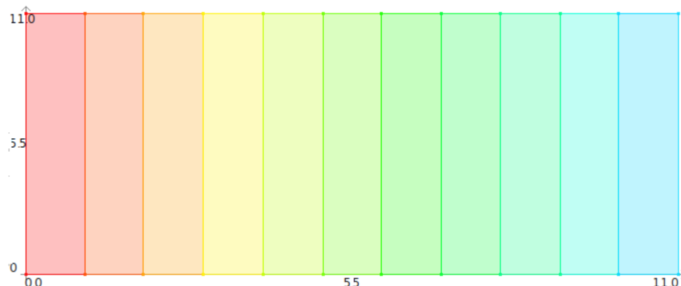

Figure 4.1: Plot showing the domains covered by Breadth First.

4.1.1 Breadth First, Depth First and Random Execution

These three algorithms are structural in nature. So, for the finite number of domains, they will be producing almost all the domains. Since these are not probabilistic, no operational profile is required. Table 4.2 shows the results produced for Breadth First Execution with this program. The program is run with minimum and maximum values: 0 and 11 respectively. We can see

Number	Domains	Integral	Cumulative
1	$0 \leq x \wedge 1.00000 > x$	0.00919	0.00919
2	$1.00000 \leq x \wedge 2.00000 > x$	0.02781	0.03700
3	$2.00000 \leq x \wedge 3.00000 > x$	0.06557	0.10257
4	$3.00000 \leq x \wedge 4.00000 > x$	0.12094	0.22351
5	$4.00000 \leq x \wedge 5.00000 > x$	0.17461	0.39812
6	$5.00000 \leq x \wedge 6.00000 > x$	0.19741	0.59553
7	$6.00000 \leq x \wedge 7.00000 > x$	0.17472	0.77025
8	$7.00000 \leq x \wedge 8.00000 > x$	0.12083	0.89108
9	$8.00000 \leq x \wedge 9.00000 > x$	0.06570	0.95678
10	$9.00000 \leq x \wedge 10.00000 > x$	0.02779	0.98457
11	$10.00000 \leq x \wedge 11.00000 \geq x$	0.00926	0.99383

Table 4.2: BreadthFirstExec takes 1224 milliseconds

that Breadth First generated all the 11 domains in ascending order of the domains and the domains produced are the expected ones, which we can see in Table 4.1.

Figure 4.1 shows the plotted domains covered by Breadth First Execution. We can clearly see the ascending order of the domains in the plot.

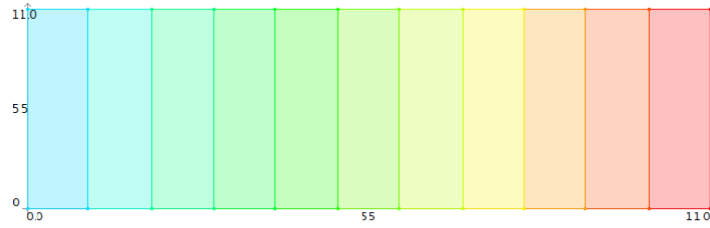

Figure 4.2: Plot showing the domains covered by Depth First.

Table 4.3 shows the results produced by Depth First Execution. We can see that this

Number	Domains	Integral	Cumulative
1	$10.00000 \leq x \wedge 11.00000 \geq x$	0.00920	0.00920
2	$9.00000 \leq x \wedge 10.00000 > x$	0.02784	0.03704
3	$8.00000 \leq x \wedge 9.00000 > x$	0.06554	0.10259
4	$7.00000 \leq x \wedge 8.00000 > x$	0.12081	0.22340
5	$6.00000 \leq x \wedge 7.00000 > x$	0.17483	0.39823
6	$5.00000 \leq x \wedge 6.00000 > x$	0.19740	0.59563
7	$4.00000 \leq x \wedge 5.00000 > x$	0.17470	0.77033
8	$3.00000 \leq x \wedge 4.00000 > x$	0.12101	0.89134
9	$2.00000 \leq x \wedge 3.00000 > x$	0.06567	0.95701
10	$1.00000 \leq x \wedge 2.00000 > x$	0.02779	0.98480
11	$0 \leq x \wedge 1.00000 > x$	0.00918	0.99397

Table 4.3: DepthFirstExec takes 1114 milliseconds

algorithm is also producing all 11 domains in descending order based on the structural nature of the algorithms.

Figure 4.2 shows the orders of the domains covered by Depth First. The plot clearly shows the descending order of the domains covered.

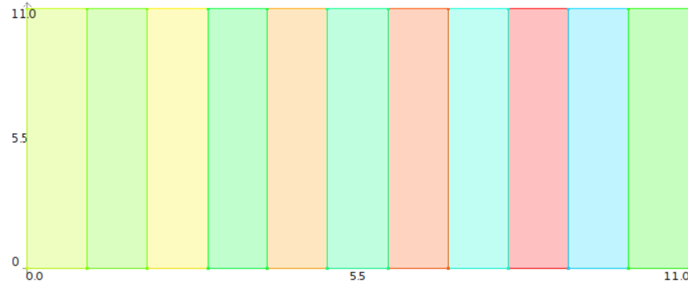

Figure 4.3: Plot showing the domains covered by Random Execution.

Table 4.4 shows the results produced by Random Execution for this finite program. We can

Number	Domains	Integral	Cumulative
1	$8.00000 \leq x \wedge 9.00000 > x$	0.06542	0.06542
2	$6.00000 \leq x \wedge 7.00000 > x$	0.17470	0.24011
3	$4.00000 \leq x \wedge 5.00000 > x$	0.17483	0.41494
4	$2.00000 \leq x \wedge 3.00000 > x$	0.06546	0.48039
5	$0 \leq x \wedge 1.00000 > x$	0.00916	0.48955
6	$1.00000 \leq x \wedge 2.00000 > x$	0.02788	0.51743
7	$10.00000 \leq x \wedge 11.00000 \geq x$	0.00923	0.52666
8	$3.00000 \leq x \wedge 4.00000 > x$	0.12091	0.64757
9	$5.00000 \leq x \wedge 6.00000 > x$	0.19738	0.84495
10	$7.00000 \leq x \wedge 8.00000 > x$	0.12099	0.96594
11	$9.00000 \leq x \wedge 10.00000 > x$	0.02770	0.99365

Table 4.4: RandomExec takes 1125 milliseconds

see that Random Execution is also producing all the domains, but the orders are just random. Figure 4.3 shows the orders of the domains covered by Random Execution. This example shows that the domains produced by Random Exec are neither in ascending nor descending orders but random because this does not flow either towards the breadth or depth but randomly choose one of them. The cumulative integrals for three of them show the same accuracy in the end.

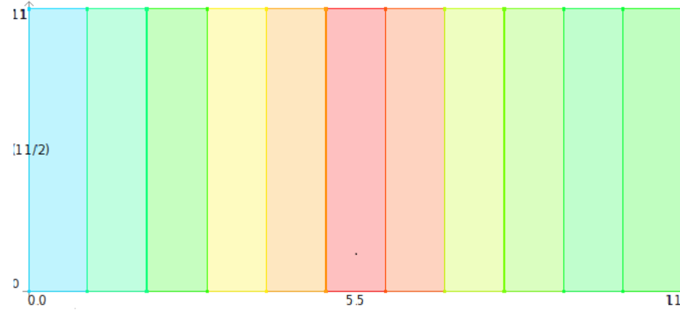

Figure 4.4: Plot showing the domains covered by Probabilistic Execution.

4.1.2 Probabilistic and Monte Carlo Execution

Both these algorithms use probability density function as an operational profile. These are run using one dimensional Normal and Weibull PDFs implemented in our experiments, whose results are discussed in the next paragraph.

Normal Distribution

This simple program is run with PDF Normal centered at 5.5 for both Probabilistic and Monte Carlo execution. The lower and upper bound is set to 0 and 11 respectively. The results of Probabilistic Execution with Normal is tabulated in Table 4.5.

Number	Domains	Integral	Cumulative
1	$5.00000 \leq x \wedge 6.00000 > x$	0.19745	0.19745
2	$4.00000 \leq x \wedge 5.00000 > x$	0.17467	0.37212
3	$6.00000 \leq x \wedge 7.00000 > x$	0.17468	0.54680
4	$7.00000 \leq x \wedge 8.00000 > x$	0.12121	0.66801
5	$3.00000 \leq x \wedge 4.00000 > x$	0.12095	0.78896
6	$8.00000 \leq x \wedge 9.00000 > x$	0.06553	0.85449
7	$2.00000 \leq x \wedge 3.00000 > x$	0.06555	0.92004
8	$9.00000 \leq x \wedge 10.00000 > x$	0.02780	0.94784
9	$1.00000 \leq x \wedge 2.00000 > x$	0.02780	0.97563
10	$10.00000 \leq x \wedge 11.00000 \geq x$	0.00928	0.98491
11	$0 \leq x \wedge 1.00000 > x$	0.00921	0.99412

Table 4.5: ProbabilisticExec with PDFNormal takes 13847 milliseconds

Figure 4.4 shows the domains covered by Probabilistic Execution. We can see that the Probabilistic Execution is displaying all 11 domains. The result shows accuracy because the first domain is the most frequent one and so on.

Figure 4.5: Plot showing the domains covered by Monte Carlo Execution.

Table 4.6 shows the result of Monte Carlo Execution with one dimensional Normal PDF. Monte Carlo Execution is also generating all 11 domains and the first domain produced is the

Number	Domains	Integral	Cumulative
1	$5.00000 \leq x \wedge 6.00000 > x$	0.19736	0.19736
2	$6.00000 \leq x \wedge 7.00000 > x$	0.17456	0.37192
3	$9.00000 \leq x \wedge 10.00000 > x$	0.02790	0.39983
4	$2.00000 \leq x \wedge 3.00000 > x$	0.06545	0.46527
5	$1.00000 \leq x \wedge 2.00000 > x$	0.02776	0.49303
6	$3.00000 \leq x \wedge 4.00000 > x$	0.12099	0.61402
7	$7.00000 \leq x \wedge 8.00000 > x$	0.12114	0.73516
8	$8.00000 \leq x \wedge 9.00000 > x$	0.06562	0.80078
9	$4.00000 \leq x \wedge 5.00000 > x$	0.17459	0.97537
10	$0 \leq x \wedge 1.00000 > x$	0.00919	0.98456
11	$10.00000 \leq x \wedge 11.00000 \geq x$	0.00917	0.99373

Table 4.6: MonteCarloExec with PDFNormal takes 1106 milliseconds

most significant one.

Figure 4.5 shows the domains covered by Monte Carlo Execution. We can see that the second domain produced is also the second significant. But all the domains produced are not in the right order which is from most to least significant which is not totally the case with Monte Carlo Execution. But because of continuations and integral calculation at each point, Probabilistic Execution is taking lot more time than Monte Carlo. So we can say that Monte Carlo generates better results for this program since the domains generated by the algorithm are somewhat significant.

Weibull Distribution

The experiment is run on this simple program with Weibull distribution whose shape parameter is set to 2. The lower and upper bounds are set to -5 and 10 respectively. The results of the Probabilistic Exec with PDF Weibull is shown in Table 4.7. This example generates 9 domains instead of 11 because of the probabilistic nature of the algorithm and they are all in right order

Number	Domains	Integral	Cumulative
1	$0 \leq x \wedge 1.00000 > x$	0.39349	0.39349
2	$1.00000 \leq x \wedge 2.00000 > x$	0.23862	0.63211
3	$2.00000 \leq x \wedge 3.00000 > x$	0.14485	0.77697
4	$3.00000 \leq x \wedge 4.00000 > x$	0.08771	0.86468
5	$4.00000 \leq x \wedge 5.00000 > x$	0.05331	0.91799
6	$5.00000 \leq x \wedge 6.00000 > x$	0.03232	0.95030
7	$6.00000 \leq x \wedge 7.00000 > x$	0.01958	0.96989
8	$8.00000 \leq x \wedge 9.00000 > x$	0.00716	0.97704
9	$9.00000 \leq x \wedge 10.00000 > x$	0.00430	0.98135

Table 4.7: ProbabilisticExec with PDFWeibull takes 19399 milliseconds

from most to least significant. Table 4.8 shows the result of Monte Carlo Execution with PDF Weibull. The generated domains are not in the order from most to least significant since the

Number	Domains	Integral	Cumulative
1	$2.00000 \leq x \wedge 3.00000 > x$	0.14475	0.14475
2	$1.00000 \leq x \wedge 2.00000 > x$	0.23862	0.38337
3	$3.00000 \leq x \wedge 4.00000 > x$	0.08784	0.47121
4	$5.00000 \leq x \wedge 6.00000 > x$	0.03229	0.50350
5	$0 \leq x \wedge 1.00000 > x$	0.39346	0.89696
6	$6.00000 \leq x \wedge 7.00000 > x$	0.01964	0.91660
7	$4.00000 \leq x \wedge 5.00000 > x$	0.05322	0.96981
8	$9.00000 \leq x \wedge 10.00000 > x$	0.00406	0.97387
9	$8.00000 \leq x \wedge 9.00000 > x$	0.00734	0.98121
10	$7.00000 \leq x \wedge 8.00000 > x$	0.01190	0.99311

Table 4.8: MonteCarloExec with PDFWeibull takes 3284 milliseconds

fifth domain is the most significant.

We can see that the first domain produced by Probabilistic Execution with both normal and weibull distribution is the most frequent domain and so on. The domains produced by Monte Carlo with Normal are more significant than weibull distribution. Probabilistic execution always takes more time than Monte Carlo.

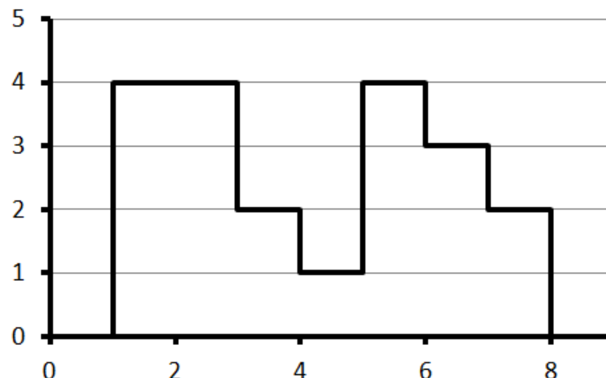

Figure 4.6: Graph of one dimensional histogram

Histograms

Both Probabilistic and Monte Carlo execution algorithms find domains from finite program using histogram as an operational profile. Histogram covers the range from 1 to 11 with highest probabilities in the range from 1 to 4. Figure 4.6 shows a graph of histogram.

Table 4.9 shows the result of Probabilistic Execution with PDF Histogram. We can see

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 2.00000 > x$	0.14282	0.14282
2	$2.00000 \leq x \wedge 3.00000 > x$	0.14276	0.28557
3	$3.00000 \leq x \wedge 4.00000 > x$	0.14272	0.42830
4	$9.00000 \leq x \wedge 10.00000 > x$	0.10714	0.53543
5	$8.00000 \leq x \wedge 9.00000 > x$	0.10713	0.64257
6	$4.00000 \leq x \wedge 5.00000 > x$	0.07154	0.71410
7	$7.00000 \leq x \wedge 8.00000 > x$	0.07149	0.78560
8	$10.00000 \leq x \wedge 11.00000 \geq x$	0.07149	0.85708
9	$5.00000 \leq x \wedge 6.00000 > x$	0.07148	0.92857
10	$6.00000 \leq x \wedge 7.00000 > x$	0.07144	1.00000

Table 4.9: ProbabilisticExec with PDFHistogram takes 10070 milliseconds

that Probabilistic execution algorithm is producing domains in the order from most to least significant. Histogram has the highest probability of occurrence from 1 to 4, second probability from 8 to 10, third probability from 4 to 5, 7 to 8 and 10 to 11, and fourth probability from 5 to 7 which is the least probability. All the domains generated by this probabilistic execution are in the right order.

Table 4.10 shows the result of Monte Carlo Execution with PDF Histogram.

Number	Domains	Integral	Cumulative
1	$2.00000 \leq x \wedge 3.00000 > x$	0.14276	0.14276
2	$9.00000 \leq x \wedge 10.00000 > x$	0.10714	0.24990
3	$10.00000 \leq x \wedge 11.00000 \geq x$	0.07149	0.32138
4	$7.00000 \leq x \wedge 8.00000 > x$	0.07149	0.39288
5	$4.00000 \leq x \wedge 5.00000 > x$	0.07154	0.46441
6	$3.00000 \leq x \wedge 4.00000 > x$	0.14272	0.60713
7	$1.00000 \leq x \wedge 2.00000 > x$	0.14282	0.74995
8	$6.00000 \leq x \wedge 7.00000 > x$	0.07144	0.82139
9	$8.00000 \leq x \wedge 9.00000 > x$	0.10713	0.92852
10	$5.00000 \leq x \wedge 6.00000 > x$	0.07148	1.00000

Table 4.10: MonteCarloExec with PDFHistogram takes 773 milliseconds

Monte Carlo Execution is not producing the domains in the right order except the first one. Although the cumulative integral for both algorithms is same but Probabilistic execution is producing domains in the right order. The time taken by Probabilistic execution is lot more than Monte Carlo execution.

4.2 An Artificial Infinite Program

An infinite program as described in the previous chapter is used to run all the five algorithms. Since Breadth First, Depth First and Random Execution algorithms do not use operational profile, they are not run with any of the pdfs. Probabilistic and Monte Carlo executions are run with both uniform and normal distributions.

4.2.1 Breadth First, Depth First and Random Execution

Table 4.11 shows the result produced by Breadth First with this program. The algorithm is

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 5.00000 > x$	0.40000	0.40000
2	$10.88235 \leq x \wedge 11.00000 \geq x$	0.01176	0.41176
3	$10.26316 \leq x \wedge 10.88235 > x$	0.06192	0.47368
4	$9.76190 \leq x \wedge 10.26316 > x$	0.05013	0.52381
5	$9.34783 \leq x \wedge 9.76190 > x$	0.04141	0.56522
6	$9.00000 < x \wedge 9.34783 > x$	0.03478	0.60000
7	$9.00000 = x$	0.00000	0.60000
8	$8.70370 \leq x \wedge 9.00000 > x$	0.02963	0.62963
9	$8.44828 \leq x \wedge 8.70370 > x$	0.02554	0.65517
10	$8.22581 \leq x \wedge 8.44828 > x$	0.02225	0.67742
...			
13	$7.70270 \leq x \wedge 7.85714 > x$	0.01544	0.72973
...			
21	$6.96078 \leq x \wedge 7.00000 \geq x$	0.00392	0.80392
...			

Table 4.11: BreadthFirstExec takes 85 milliseconds

producing more than 20 domains but we cut off the domains at 21. The integral for each of the domains is not the necessary ones. Figure 4.7 shows the domains covered by Breadth First Execution. The minimum and the maximum values for the experiment are 1.0 and 11.0

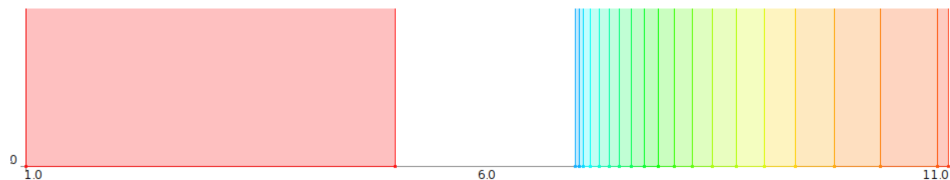

Figure 4.7: Plot showing the domains covered by Breadth First.

respectively. The coloured area shows the covered domains whereas the white area within the input space shows no domain produced. The order of the domains starts from dark red, then becomes lighter and lighter and ends at blue.

Table 4.12 shows the result produced by Depth First Execution. The domains produced

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 5.00000 > x$	0.40000	0.40000
2	$6.96078 \leq x \wedge 7.00000 \geq x$	0.00392	0.40392
3	$6.88679 \leq x \wedge 6.96078 > x$	0.00740	0.41132
4	$6.81818 \leq x \wedge 6.88679 > x$	0.00686	0.41818
5	$6.75439 \leq x \wedge 6.81818 > x$	0.00638	0.42456
6	$6.69492 \leq x \wedge 6.75439 > x$	0.00595	0.43051
7	$6.63934 \leq x \wedge 6.69492 > x$	0.00556	0.43607
8	$6.58730 \leq x \wedge 6.63934 > x$	0.00520	0.44127
9	$6.53846 \leq x \wedge 6.58730 > x$	0.00488	0.44615
10	$6.49254 \leq x \wedge 6.53846 > x$	0.00459	0.45075
	...		
13	$6.36986 \leq x \wedge 6.40845 > x$	0.00386	0.46301
	...		
21	$6.12360 \leq x \wedge 6.14943 > x$	0.00258	0.48764
	...		

Table 4.12: DepthFirstExec takes 76 milliseconds

by Depth first are also not the necessary ones because the integral is very small. Number of domains produced by Depth First Execution is also more than 20 but we are cutting off them at 21. Figure 4.8 shows the domains covered by Depth First Execution. The coloured area

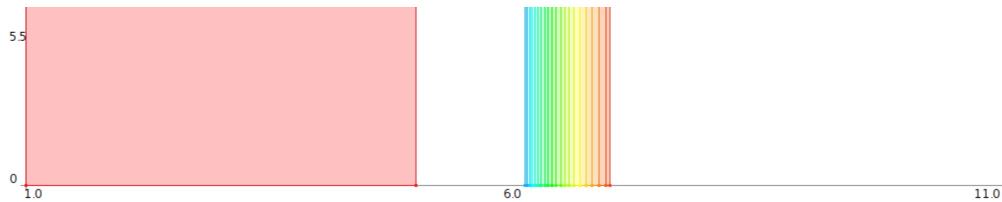

Figure 4.8: Plot showing the domains covered by Depth First.

shows the covered domains and the white area shows no domain. The domains are produced as the structural nature of the algorithm. Different range of colours show the order in which the domains are produced.

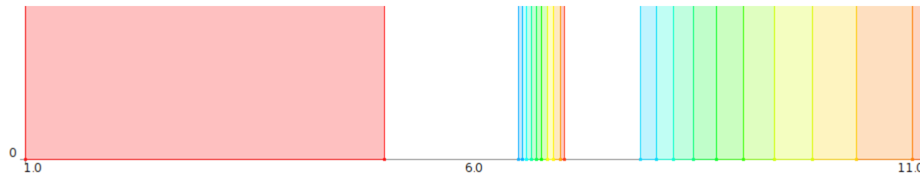

Figure 4.9: Plot showing the domains covered by Random Execution

Table 4.13 shows the result produced by Random Execution. The domains produced for

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 5.00000 > x$	0.40000	0.40000
2	$6.96078 \leq x \wedge 7.00000 \geq x$	0.00392	0.40392
3	$10.88235 \leq x \wedge 11.00000 \geq x$	0.01176	0.41569
4	$10.26316 \leq x \wedge 10.88235 > x$	0.06192	0.47761
5	$6.88679 \leq x \wedge 6.96078 > x$	0.00740	0.48500
6	$9.76190 \leq x \wedge 10.26316 > x$	0.05013	0.53513
7	$6.81818 \leq x \wedge 6.88679 > x$	0.00686	0.54199
8	$9.34783 \leq x \wedge 9.76190 > x$	0.04141	0.58340
9	$9.00000 = x$	0.00000	0.58340
10	$9.00000 < x \wedge 9.34783 > x$	0.03478	0.61818
...			
13	$6.69492 \leq x \wedge 6.75439 > x$	0.00595	0.66014
...			
21	$6.49254 \leq x \wedge 6.53846 > x$	0.00459	0.76503
...			

Table 4.13: RandomExec takes 84 milliseconds

all three algorithms have different sequence. It is shown that both Breadth First and Depth First take almost the same time and the cumulative integral for Random Exec is better than the both Breadth First and Depth First. Figure 4.9 shows the domains covered by Random Execution. We can see that domains produced by Random Execution do not have a specific order like breadth first and depth first since the algorithm alternates between the two.

4.2.2 Probabilistic and Monte Carlo Execution

Both algorithms are run with this infinite program using both uniform and normal distributions.

Uniform Distribution

Minimum and maximum values for this distribution are 1.0 and 11.0 respectively. The domains produced for Probabilistic Execution with PDF Uniform is shown in Table 4.14. The algorithm

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 5.00000 > x$	0.40000	0.40000
2	$10.26316 \leq x \wedge 10.88235 > x$	0.06192	0.46192
3	$9.76190 \leq x \wedge 10.26316 > x$	0.05013	0.51204
4	$9.34783 \leq x \wedge 9.76190 > x$	0.04141	0.55345
5	$9.00000 < x \wedge 9.34783 > x$	0.03478	0.58824
6	$8.70370 \leq x \wedge 9.00000 > x$	0.02963	0.61786
7	$8.44828 \leq x \wedge 8.70370 > x$	0.02554	0.64341
8	$8.22581 \leq x \wedge 8.44828 > x$	0.02225	0.66565
9	$8.03030 \leq x \wedge 8.22581 > x$	0.01955	0.68520
10	$7.85714 \leq x \wedge 8.03030 > x$	0.01732	0.70252
	...		
13	$7.43902 \leq x \wedge 7.56410 > x$	0.01251	0.74433
	...		
21	$6.75439 \leq x \wedge 6.81818 > x$	0.00638	0.81656

Table 4.14: ProbabilisticExec with PDFUniform takes 1607 milliseconds

produced 21 domains and all the domains are produced in the right order. Although the integral of each domain except the first one is small but the total integral gives the impression of accuracy.

Figure 4.10 shows the domain produced by Probabilistic Execution. Coloured lines show

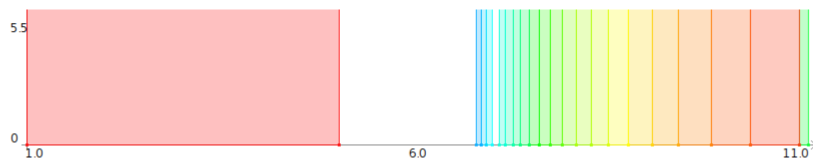

Figure 4.10: Plot showing the domains covered by Probabilistic Execution

the domains produced whereas white area shows no domain. The domains are shown as the probabilistic nature of the algorithm. We can see that there is a small gap between the domains covered in the plot which means the algorithm is not covering a small probability domain. The reason for this is that the RSS algorithm is producing 0 integral for this small domain.

Table 4.15 shows the results produced by Monte Carlo Execution. The order of the domains

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 5.00000 > x$	0.40000	0.40000
2	$9.34783 \leq x \wedge 9.76190 > x$	0.04141	0.44141
3	$6.49254 \leq x \wedge 6.53846 > x$	0.00459	0.44600
4	$9.00000 < x \wedge 9.34783 > x$	0.03478	0.48078
5	$7.85714 \leq x \wedge 8.03030 > x$	0.01732	0.49810
6	$8.70370 \leq x \wedge 9.00000 > x$	0.02963	0.52773
7	$9.76190 \leq x \wedge 10.26316 > x$	0.05013	0.57785
8	$8.22581 \leq x \wedge 8.44828 > x$	0.02225	0.60010
9	$7.70270 \leq x \wedge 7.85714 > x$	0.01544	0.61554
10	$6.81818 \leq x \wedge 6.88679 > x$	0.00686	0.62241
...			
13	$7.43902 \leq x \wedge 7.56410 > x$	0.01251	0.70204
...			
21	$10.88235 \leq x \wedge 11.00000 \geq x$	0.01176	0.77463

Table 4.15: MonteCarloExec with PDFUniform takes 401 milliseconds

generated shows that they also have the most frequent domains first. The total integral is smaller than the Probabilistic one.

Figure 4.11 shows the plot of Monte Carlo Execution. Since Probabilistic Execution captures

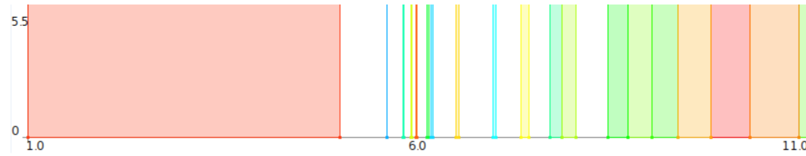

Figure 4.11: Plot showing the domains covered by Monte Carlo Execution

the continuation at the point whenever a probabilistic predicate has to be evaluated, it takes more time and produces the most frequent domains first. The most frequent domain has the highest value of the integral as we can see in table.

Normal Distribution

Both algorithms are also run with PDF Normal centered at 5.

Table 4.16 shows the results produced by Probabilistic Execution. This test shows that

Number	Domains	Integral	Cumulative
1	$1.00000 \leq x \wedge 5.00000 > x$	0.15873	0.15873
2	$7.22222 \leq x \wedge 7.32558 > x$	0.01832	0.17705
3	$7.56410 \leq x \wedge 7.70270 > x$	0.01458	0.19163
4	$7.70270 \leq x \wedge 7.85714 > x$	0.01267	0.20430
5	$7.85714 \leq x \wedge 8.03030 > x$	0.01056	0.21485
6	$8.03030 \leq x \wedge 8.22581 > x$	0.00810	0.22296
7	$8.22581 \leq x \wedge 8.44828 > x$	0.00579	0.22875
8	$8.70370 \leq x \wedge 9.00000 > x$	0.00209	0.23084
9	$9.00000 < x \wedge 9.34783 > x$	0.00108	0.23192
10	$9.34783 \leq x \wedge 9.76190 > x$	0.00020	0.23211
...			
13	$10.88235 \leq x \wedge 11.00000 \geq x$	0.00000	0.23221

Table 4.16: ProbabilisticExec with PDFNormal takes 107978 milliseconds

Probabilistic Execution took a long time to produce the domains and also integral of the domains is small.

Table 4.17 shows the results produced by Monte Carlo Execution. Monte Carlo Execution

Number	Domains	Integral	Cumulative
1	$5.62893 \leq x \wedge 5.63694 > x$	0.00000	0.00000
2	$7.22222 \leq x \wedge 7.32558 > x$	0.01834	0.01834
3	$5.38911 \leq x \wedge 5.39216 > x$	0.00000	0.01834
4	$6.53846 \leq x \wedge 6.58730 > x$	0.00000	0.01834
5	$5.12063 \leq x \wedge 5.12092 > x$	0.00000	0.01834
6	$1.00000 \leq x \wedge 5.00000 > x$	0.15887	0.17720
7	$6.07527 \leq x \wedge 6.09890 > x$	0.00939	0.18659
8	$7.04082 \leq x \wedge 7.12766 > x$	0.00000	0.18659
9	$6.49254 \leq x \wedge 6.53846 > x$	0.00000	0.18659
10	$5.60606 \leq x \wedge 5.61350 > x$	0.00000	0.18659
...			
13	$6.17647 \leq x \wedge 6.20482 > x$	0.00000	0.18659
...			
21	$8.44828 \leq x \wedge 8.70370 > x$	0.00366	0.19992

Table 4.17: MonteCarloExec with PDFNormal takes 1197 milliseconds

produces the domains which are not in the right order. We can see that probabilistic execution produces only 13 domains whereas Monte Carlo execution produces 21 domains in very less time. Probabilistic Execution is producing less domains because it does the integral calculation to check if the domain is significant. So it does not produce insignificant domains.

4.3 Sine

We ran tests on all the five algorithms with the Sine program. See section (3.4.3) for Sine. Tests are run with both normal and weibull distributions. These are both one dimensional pdfs used with sine function.

4.3.1 Breadth First, Depth First and Random Execution

As mentioned earlier, these algorithms do not use operational profiles, so they do not require probability density functions, which are used as operational profiles. The minimum and maximum values used for these runs are -10 *pi* and 10 *pi* respectively.

Table 4.18 shows the results produced by Breadth First Execution. We can see that Breadth

Number	Domains	Integral	Cumulative
1	$6.28319 < x \wedge 31.41593 \geq x$	0.33646	0.33646
2	$-31.41593 \leq x \wedge -6.28319 > x$	0.30289	0.63935
3	$0 \leq x \wedge 1.0e^{-12} \geq x$	0.00000	0.63935
4	$-1.0e^{-12} \leq x \wedge 0 > x$	0.00000	0.63935
5	$1.2e^{-10} \geq x^5 \wedge 1.0e^{-12} < x \wedge 1.57080 \geq x$	0.00000	0.63935
6	$-1.2e^{-10} \leq x^5 \wedge -1.57080 \leq x \wedge -1.0e^{-12} > x$	0.00000	0.63935
7	$3.62880e^{-7} \geq x^9 \wedge 1.2e^{-10} < x^5 \wedge 1.0e^{-12} < x \wedge 1.57080 \geq x \wedge \cdots$	0.00484	0.64419
8	$-3.62880e^{-7} \leq x^9 \wedge -1.2e^{-10} > x^5 \wedge -1.57080 \leq x \wedge -1.0e^{-12} > x \wedge \cdots$	0.00000	0.64419
9	$3.14159 < x \wedge 3.14159 \geq x$	0.00000	0.64419
10	$0.00623 \geq x^{13} \wedge 3.62880e^{-7} < x^9 \wedge 1.2e^{-10} < x^5 \wedge 1.0e^{-12} < x \wedge \cdots$	0.01287	0.65706
$\cdots$			
13	$306.01968 \geq x^5 - 15.70796x^4 + 98.69604x^3 + \cdots \wedge 3.14159 < x \wedge 4.71239 \geq x$	0.00000	0.65706
$\cdots$			
21	$5.10909e^7 \geq x^{21} \wedge 355.68743 < x^{17} \wedge 0.00623 < x^{13} \wedge 3.62880e^{-7} < x^9 \wedge \cdots$	0.00000	0.67649

Table 4.18: BreadthFirstExec takes 137 milliseconds

first is producing some useful domains because the algorithm just happens to find them.

Table 4.19 shows the results produced by Depth First Execution. We can see that Depth

Number	Domains	Integral	Cumulative
1	$0 \leq x \wedge 1.0e^{-12} \geq x$	0.00000	0.00000
2	$1.2e^{-10} \geq x^5 \wedge 1.0e^{-12} < x \wedge 1.57080 \geq x$	0.00000	0.00000
3	$3.62880e^{-7} \geq x^9 \wedge 1.2e^{-10} < x^5 \wedge 1.0e^{-12} < x \wedge 1.57080 \geq x \wedge \ldots$	0.00000	0.00000
4	$0.00623 \geq x^{13} \wedge 3.62880e^{-7} < x^9 \wedge 1.2e^{-10} < x^5 \wedge 1.0e^{-12} < x \wedge \cdots$	0.01284	0.01284
5	$355.68743 \geq x^{17} \wedge 0.00623 < x^{13} \wedge 3.62880e^{-7} < x^9 \wedge 1.2e^{-10} < x^5 \wedge \cdots$	0.01957	0.03242
6	$5.10909e^7 \geq x^{21} \wedge 355.68743 < x^{17} \wedge 0.00623 < x^{13} \wedge 3.62880e^{-7} < x^9 \wedge \cdots$	0.00420	0.03662
7	$1.55112e^{13} \geq x^{25} \wedge 5.10909e^7 < x^{21} \wedge 355.68743 < x^{17} \wedge 0.00623 < x^{13} \wedge \cdots$	0.00000	0.03662
8	$8.84176e^{18} \geq x^{29} \wedge 1.55112e^{13} < x^{25} \wedge 5.10909e^7 < x^{21} \wedge 355.68743 < x^{17} \wedge \cdots$	0.00000	0.03662
9	$8.68331e^{24} \geq x^{33} \wedge 8.84176e^{18} < x^{29} \wedge 1.55112e^{13} < x^{25} \wedge 5.10909e^7 < x^{21} \wedge \cdots$	0.00000	0.03662
10	$1.37637e^{31} \geq x^{37} \wedge 8.68331e^{24} < x^{33} \wedge 8.84176e^{18} < x^{29} \wedge 1.55112e^{13} < x^{25} \wedge \cdots$	0.00000	0.03662
	$\cdots$		
13	$6.08281e^{50} \geq x^{49} \wedge 1.19622e^{44} < x^{45} \wedge 3.34525e^{37} < x^{41} \wedge 1.37637e^{31} < x^{37} \wedge \cdots$	0.00000	0.03662
	$\cdots$		
22	$2.81710e^{116} \geq x^{85} \wedge 5.79712e^{108} < x^{81} \wedge 1.45183e^{101} < x^{77} \wedge 4.47011e^{93} < x^{73} \wedge \cdots$	0.00000	0.03662
	$\cdots$		
41	$7.59070e^{274} \geq x^{161} \wedge 1.17295e^{266} < x^{157} \wedge 2.00634e^{257} < x^{153} \wedge 3.80892e^{248} < x^{149} \wedge \cdots$	0.00000	0.03662

Table 4.19: DepthFirstExec takes 740 milliseconds

First Execution for this run does not produce any useful domain. The algorithm does not happen to find any useful domain because of its structural nature.

Table 4.20 shows the results produced by Random Execution. The tabular result shows

Number	Domains	Integral	Cumulative
1	$6.28319 < x \wedge 31.41593 \geq x$	0.33622	0.33622
2	$-31.41593 \leq x \wedge -6.28319 > x$	0.30326	0.63948
3	$0 \leq x \wedge 1.0e^{-12} \geq x$	0.00000	0.63948
4	$-1.0e^{-12} \leq x \wedge 0 > x$	0.00000	0.63948
5	$1.2e^{-10} \geq x^5 \wedge 1.0e^{-12} < x \wedge 1.57080 \geq x$	0.00000	0.63948
6	$306.01968 \geq x^5 - 15.70796x^4 + 98.69604x^3 + \cdots \wedge 3.14159 < x \wedge 4.71239 \geq x$	0.00000	0.63948
7	$-1.2e^{-10} \leq x^5 \wedge -1.57080 \leq x \wedge -1.0e^{-12} > x$	0.00000	0.63948
8	$-306.01968 \leq x^5 + 15.70796x^4 + 98.69604x^3 + \cdots \wedge -4.71239 \leq x \wedge -3.14159 > x$	0.00000	0.63948
9	$3.62880e^{-7} \geq x^9 \wedge 1.2e^{-10} < x^5 \wedge 1.0e^{-12} < x \wedge 1.57080 \geq x \wedge \cdots$	0.00484	0.64432
10	$3.14159 < x \wedge 3.14159 \geq x$	0.00000	0.64432
	$\cdots$		
13	$-3.14159 \leq x \wedge -3.14159 > x$	0.00000	0.66182
	$\cdots$		
21	$-29809.09933 \leq x^9 + 28.27433x^8 + 355.30576x^7 + \cdots \wedge -306.01968 > x^5 + 15.70796x^4 + 98.69604x^3 + \cdots \wedge -4.71239 \leq x \wedge -3.14159 > x \wedge \cdots$	0.00468	0.70861

Table 4.20: RandomExec takes 661 milliseconds

that Random Execution is producing some significant domains and has the same cumulative integral as the Breadth First Execution.

4.3.2 Probabilistic and Monte Carlo Execution

Both algorithms were run using normal distribution as a pdf. The min and max values used for the runs were -10 *pi* and 10 *pi* respectively. The normal distribution used for the tests have mean centered at *pi*/4.

Table 4.21 shows the results of the experiments for Probabilistic Execution. We can see

Number	Domains	Integral	Cumulative
1	$6.28319 < x \wedge 31.41593 \geq x$	0.33613	0.33613
2	$-31.41593 \leq x \wedge -6.28319 > x$	0.30278	0.63890
3	$-3.70730e^{13} \geq x^{17} + 106.81415x^{16} + 5369.06479x^{15} + \cdots \wedge$ $-2.37869e^{10} < x^{13} + 81.68141x^{12} + 3079.31657x^{11} + \cdots \wedge$ $-1.52622e^{7} < x^{9} + 56.54867x^{8} + 1421.22303x^{7} + \cdots \wedge$ $-9792.62991 < x^{5} + 31.41593x^{4} + 394.78418x^{3} + \cdots \wedge \cdots$	0.03271	0.67162
4	$-355.68743 \leq x^{17} \wedge -0.00623 > x^{13} \wedge -3.62880e^{-7} > x^{9} \wedge$ $-1.2e^{-10} > x^{5} \wedge \cdots$	0.01945	0.69107
5	$-2.82844e^{8} \geq x^{17} + 53.40708x^{16} + 1342.26620x^{15} + \cdots \wedge$ $-2.90367e^{6} < x^{13} + 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $-29809.09933 < x^{9} + 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $-306.01968 < x^{5} + 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge \cdots$	0.01922	0.71029
6	$-2.82844e^{8} \leq x^{17} + 53.40708x^{16} + 1342.26620x^{15} + \cdots \wedge$ $-2.90367e^{6} > x^{13} + 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $-29809.09933 > x^{9} + 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $-306.01968 > x^{5} + 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge \cdots$	0.01852	0.72882
7	$0.00623 \geq x^{13} \wedge 3.62880e^{-7} < x^{9} \wedge 1.2e^{-10} < x^{5} \wedge$ $1.0e^{-12} < x \wedge \cdots$	0.00000	0.72882
8	$-0.00623 \leq x^{13} \wedge -3.62880e^{-7} > x^{9} \wedge -1.2e^{-10} > x^{5} \wedge$ $-1.57080 \leq x \wedge \cdots$	0.01284	0.74166
9	$2.90367e^{6} \geq x^{13} - 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $29809.09933 < x^{9} - 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $306.01968 < x^{5} - 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge$ $3.14159 < x \wedge \cdots$	0.00000	0.74166
10	$-2.90367e^{6} \leq x^{13} + 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $-29809.09933 > x^{9} + 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $-306.01968 > x^{5} + 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge$ $-4.71239 \leq x \wedge \cdots$	0.01235	0.75401
$\cdots$			
12	$-5.77799e^{16} \geq x^{21} + 131.94689x^{20} + 8290.46770x^{19} + \cdots \wedge$ $-3.70730e^{13} < x^{17} + 106.81415x^{16} + 5369.06479x^{15} + \cdots \wedge$ $-2.37869e^{10} < x^{13} + 81.68141x^{12} + 3079.31657x^{11} + \cdots \wedge$ $-1.52622e^{7} < x^{9} + 56.54867x^{8} + 1421.22303x^{7} + \cdots \wedge \cdots$	0.00000	0.75401

Table 4.21: ProbabilisticExec with PDFNormal takes 10919 milliseconds

that the algorithm provide the accurate results but time taken to produce the results is very long.

Table 4.22 shows the results of the experiments for Monte Carlo Execution. This also provides the same accuracy as Probabilistic Execution. We can see that Monte Carlo Execution produces lot more domains in less time than Probabilistic Execution.

Number	Domains	Integral	Cumulative
1	$6.28319 < x \wedge 31.41593 \geq x$	0.33646	0.33646
2	$-31.41593 \leq x \wedge -6.28319 > x$	0.30289	0.63935
3	$2.90367e^{6} \leq x^{13} - 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $29809.09933 > x^{9} - 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $306.01968 > x^{5} - 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge$ $1.57080 < x \wedge \cdots$	0.00000	0.63935
4	$2.82844e^{8} \leq x^{17} - 53.40708x^{16} + 1342.26620x^{15} + \cdots \wedge$ $2.90367e^{6} > x^{13} - 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $29809.09933 > x^{9} - 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $306.01968 > x^{5} - 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge \cdots$	0.00000	0.63935
5	$-0.00623 \leq x^{13} \wedge -3.62880e^{-7} > x^{9} \wedge -1.2e^{-10} > x^{5} \wedge$ $-1.57080 \leq x \wedge \cdots$	0.01284	0.65219
6	$-355.68743 \leq x^{17} \wedge -0.00623 > x^{13} \wedge -3.62880e^{-7} > x^{9} \wedge$ $-1.2e^{-10} > x^{5} \wedge \cdots$	0.00000	0.65219
7	$-2.90367e^{6} \leq x^{13} + 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $-29809.09933 > x^{9} + 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $-306.01968 > x^{5} + 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge$ $-4.71239 \leq x \wedge \cdots$	0.00000	0.65219
8	$2.75005e^{10} \leq x^{21} - 65.97345x^{20} + 2072.61692x^{19} + \cdots \wedge$ $2.82844e^{8} > x^{17} - 53.40708x^{16} + 1342.26620x^{15} + \cdots \wedge$ $2.90367e^{6} > x^{13} - 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $29809.09933 > x^{9} - 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge \cdots$	0.00000	0.65219
9	$355.68743 \geq x^{17} \wedge 0.00623 < x^{13} \wedge 3.62880e^{-7} < x^{9} \wedge$ $1.2e^{-10} < x^{5} \wedge \cdots$	0.01957	0.67176
10	$2.82844e^{8} \geq x^{17} - 53.40708x^{16} + 1342.26620x^{15} + \cdots \wedge$ $2.90367e^{6} < x^{13} - 40.84070x^{12} + 769.82914x^{11} + \cdots \wedge$ $29809.09933 < x^{9} - 28.27433x^{8} + 355.30576x^{7} + \cdots \wedge$ $306.01968 < x^{5} - 15.70796x^{4} + 98.69604x^{3} + \cdots \wedge \cdots$	0.00000	0.67176
$\cdots$			
13	$0.00623 \geq x^{13} \wedge 3.62880e^{-7} < x^{9} \wedge 1.2e^{-10} < x^{5} \wedge$ $1.0e^{-12} < x \wedge \cdots$	0.00000	0.67176
$\cdots$			
21	$3.70730e^{13} \leq x^{17} - 106.81415x^{16} + 5369.06479x^{15} + \cdots \wedge$ $2.37869e^{10} > x^{13} - 81.68141x^{12} + 3079.31657x^{11} + \cdots \wedge$ $1.52622e^{7} > x^{9} - 56.54867x^{8} + 1421.22303x^{7} + \cdots \wedge$ $9792.62991 > x^{5} - 31.41593x^{4} + 394.78418x^{3} + \cdots \wedge \cdots$	0.03438	0.82189

Table 4.22: MonteCarloExec with PDFNormal takes 3006 milliseconds

4.4 ArcTan

The two dimensional function was run using 2 dimensional histogram. See section 3.2.4 for histograms. Table 4.22 shows the result of the Breadth First Exec with this function. The algo-

Number	Domains
1	$0 = x \wedge 0 < y \wedge 12.00000 \geq y$
2	$0 = x \wedge -12.00000 \leq y \wedge 0 \geq y$
3	$-12.00000 \leq x \wedge 0 > x \wedge 0 = y$
4	$0 \geq \frac{2.46740x^2-3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2+2.00000xy+y^2} \wedge -12.00000 \leq x \wedge$ $0 > x \wedge 0 \leq xy^{-1} \wedge \cdots$
5	$0 \geq \frac{2.46740x^2+3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2-2.00000xy+y^2} \wedge -12.00000 \leq x \wedge$ $0 > x \wedge 0 > xy^{-1} \wedge \cdots$
6	$0 < \frac{2.46740x^2-3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2+2.00000xy+y^2} \wedge 0 > \frac{4.71239x+6.28319y}{x+y} \wedge$ $0 \leq \frac{1.57080x}{x+y} \wedge -12.00000 \leq x \wedge \cdots$
7	$0 < \frac{2.46740x^2+3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2-2.00000xy+y^2} \wedge 0 > \frac{4.71239x-6.28319y}{x-1.00000y} \wedge$ $0 \leq \frac{1.57080x}{x-1.00000y} \wedge -12.00000 \leq x \wedge \cdots$
8	$0 < \frac{2.46740x^2-3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2+2.00000xy+y^2} \wedge 0 > \frac{7.85398x+6.28319y}{x+y} \wedge$ $0 > \frac{1.57080x}{x+y} \wedge -12.00000 \leq x \wedge \cdots$
9	$0 < \frac{2.46740x^2+3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2-2.00000xy+y^2} \wedge 0 > \frac{7.85398x-6.28319y}{x-1.00000y} \wedge$ $0 > \frac{1.57080x}{x-1.00000y} \wedge -12.00000 \leq x \wedge \cdots$
10	$0 < \frac{2.46740x^2-3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2+2.00000xy+y^2} \wedge 0 > \frac{7.85398x+9.42478y}{x+y} \wedge$ $0 \leq \frac{4.71239x+6.28319y}{x+y} \wedge 0 > \frac{1.57080x+3.14159y}{x+y} \wedge \cdots$
$\cdots$	
13	$0 < \frac{2.46740x^2+3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2-2.00000xy+y^2} \wedge 0 \leq \frac{4.71239x-6.28319y}{x-1.00000y} \wedge$ $0 \leq \frac{1.57080x-3.14159y}{x-1.00000y} \wedge 0 \leq \frac{1.57080x}{x-1.00000y} \wedge \cdots$
$\cdots$	
22	$0 < \frac{2.46740x^2-3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2+2.00000xy+y^2} \wedge 0 \leq \frac{7.85398x+6.28319y}{x+y} \wedge$ $0 \leq \frac{4.71239x+3.14159y}{x+y} \wedge 0 > \frac{3.14159x+1.57080y}{x+y} \wedge \cdots$
$\cdots$	
26	$0 < \frac{2.46740x^2+3.14159e^{-12}xy-1.57079e^{-12}y^2}{x^2-2.00000xy+y^2} \wedge 0 \leq \frac{7.85398x-6.28319y}{x-1.00000y} \wedge$ $0 \leq \frac{4.71239x-3.14159y}{x-1.00000y} \wedge 0 \leq \frac{3.14159x-1.57080y}{x-1.00000y} \wedge \cdots$
$\cdots$	

Table 4.23: BreadthFirstExec takes 653 milliseconds

rithm generated more than 21 domains but the accuracy of the domains can not be estimated because of the huge predicates.

This two dimensional function really produces complex predicates with more than 100 clauses which the language compiler does not support. So the integral calculation was not possible because of the huge predicates generated by the function.

Chapter 5

Conclusions and Future Work

My thesis is that Probabilistic Program Execution is a viable way to find paths and domains from software.

Probabilistic execution algorithms namely Probabilistic Exec and Monte Carlo Exec algorithms produced domains more or less in the order from most to least significant. From three non- probabilistic algorithms namely: Breadth First Exec, Depth First Exec and Random Exec, Random Exec algorithm is the best one since it will eventually find all the domains but will have a mixture of significant and in-significant domains. So it may take a lot of time to find all the significant domains.

5.1 Conclusions

1. From the algorithms that ignore operational profile, both Breadth First Exec and Depth First Exec are not the practical approaches because both may run into infinite path loops and infinite path lengths.

2. The Breadth First and Depth First Exec algorithms can eventually find all finite domains but they get lost for infinite domains.

3. Random Exec algorithm is the best one that works without operational profile. This algorithm will eventually find all the domains but it will have a mixture of significant and insignificant domains. So it may take a lot of time to find all significant domains.

4. The Probabilistic Exec algorithm is theoretically clean, but implementationally difficult algorithm based on program continuations. The integral calculation is done at each point which takes more time. Lets consider the experiment with Sine in Chapter 4. See section 4.3 for the experiment. The Probabilistic Execution when run with normal

distribution, have 12 domains generated whereas Monte Carlo Execution have 21 domains. See Tables 4.16 and 4.17. This is because of the integral calculation at each point to check if the domain generated is the most frequent one and so on. So we can say that integral calculation for each domain needs a fairly accurate integral calculation algorithm.

5. The Monte Carlo Exec algorithm provides the accuracy very close to the Probabilistic Execution. They generate domains more-or-less in the order from the most to least significant, as determined by the operational profile.

6. Although the domains produced by Monte Carlo Exec may not strictly be the highest frequency domain but the first domains produced tend to be the most important.

7. So we can say that Monte Carlo Exec algorithm is an implementationally simple and inexpensive algorithm that produces the most important domains first.

5.2 Contributions

Following are the contributions of this work:

1. five execution based algorithms are studied that extract the domains from the program.

2. implementation of RSS (Recursive Stratified Sampling) algorithm in smalltalk for the approximation of the integral which gave negative result particularly for Probabilistic Execution.

3. three real probability density functions are integrated to obtain the probability that the random variable takes a value in the given integral.

4. exploration of execution based algorithms using some programs that tests the accuracy of the algorithms and time taken to produce the domains.

5.3 Limitations and Future Work

There are some limitations of the current work that can be explored in the future.

1. The RSS algorithm does not seem to provide a reasonably accurate integrals because when run for the Probabilistic Exec algorithm, it sometimes gives zero integral for several non-empty domains. This is particularly noticeable with Probabilistic Exec because it calculates the integral at each point.

2. The integral calculation becomes complex with the dimensionality of the probability density function that represents an operational profile. In our implementation, this limited our ability to calculate integrals for some very complex domains. The language compiler does not support complex domains i.e.the ones that consists of predicates with 100 clauses.

3. A possible avenue for exploration would be finding an accurate integral calculation approach.

Bibliography

[1] http://en.wikipedia.org/wiki/Normal_distribution.

[2] http://en.wikipedia.org/wiki/Weibull_distribution.

[3] http://en.wikipedia.org/wiki/Monte_Carlo_integration.

[4] http://en.wikipedia.org/wiki/Histogram.

[5] Taweesup Apiwattanapong and Mary Jean Harrold. Selective path profiling. *SIGSOFT Softw. Eng. Notes*, 28(1):35–42, November 2002.

[6] Thomas Ball and James R. Larus. Programs follow paths. Technical report, MICROSOFT RESEARCH, 1999.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[8] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Trans. Softw. Eng.*, 8(4):380–390, July 1982.

[9] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 52–58, New York, NY, USA, 2005. ACM.

[10] Bingchiang Jeng and Elaine J. Weyuker. A simplified domain-testing strategy. *ACM Trans. Softw. Eng. Methodol.*, 3(3):254–270, July 1994.

[11] Ranjit Jhala and Rupak Majumdar. Path slicing. *SIGPLAN Not.*, 40(6):38–47, June 2005.

[12] Lawrence Joseph and Caroline Reinhold. Intoduction to probability theory and sampling distributions. *American journal of Roentgenology*, pages 917–923, April 2003.

[13] D Mason and D Woit. Input domain analysis for software reliability measurement. In *Proceedings of The Fifth International Conference on Computer Science and Informatics*, Atlantic City, USA, 2000.

[14] Dave Mason. Probability density functions in program analysis, 2002.

[15] Dave Mason. Probabilistic program analysis for domain/path discovery. 2009.

[16] David Victor Mason. *Probabilistic program analysis for software component reliability.* PhD thesis, University of Waterloo, Waterloo, Ont., Canada, 2002.

[17] David Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. *SIGPLAN Not.*, 36(3):93–101, January 2001.

[18] John Musa. Developing more reliable software faster and cheaper. In *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, ICECCS '99, pages 180–, Washington, DC, USA, 1999. IEEE Computer Society.

[19] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, March 1993.

[20] Darren Ng and David R. Kaeli. Program comprehension using aspects. In *ICSE 2004 Workshop WoDiSEE*, 2004.

[21] William H. Press and Glennys R. Farrar. Recursive stratified sampling for multidimensional monte carlo integration. *Comput. Phys.*, 4(2):190–195, February 1990.

[22] Joseph R. Ruthruff, Sebastian Elbaum, and Gregg Rothermel. Experimental program analysis: a new program analysis paradigm. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 49–60, New York, NY, USA, 2006. ACM.

[23] L.J. White and E.I. Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, SE-6(3):247 – 257, May 1980.