Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

6-21-2011 12:00 AM

Models, Techniques, and Metrics for Managing Risk in Software Engineering

Andriy Miranskyy,

Supervisor: Matt Davison, *The University of Western Ontario*
: Nazim H. Madhavji, *The University of Western Ontario*
: R. Mark Reesor, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Applied Mathematics

© Andriy Miranskyy 2011

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Other Applied Mathematics Commons, and the Software Engineering Commons

Recommended Citation
Miranskyy, Andriy, "Models, Techniques, and Metrics for Managing Risk in Software Engineering" (2011). *Electronic Thesis and Dissertation Repository*. 188.
https://ir.lib.uwo.ca/etd/188

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

MODELS, TECHNIQUES, AND METRICS FOR MANAGING RISK IN SOFTWARE ENGINEERING

(Spine title: Models, Techniques, and Metrics for Managing Risk in Software Engineering)

(Thesis format: Integrated Article)

by

Andriy V. Miranskyy

Graduate Program in Applied Mathematics

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Andriy Miranskyy, 2011

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Joint-Supervisor	Examiners
______________________	______________________
Dr. Nazim H. Madhavji	Dr. Daniel M. Berry
Joint-Supervisor	______________________
	Dr. Robert M. Corless
______________________	
Dr. Mark Reesor	______________________
Joint-Supervisor	Dr. Christopher Essex
______________________	______________________
Dr. Matt Davison	Dr. Stephen M. Watt

The thesis by

Andriy Miranskyy

entitled:

Models, Techniques, and Metrics for Managing Risk in Software Engineering

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

______________________	______________________
Date	Chair of the Thesis Examination Board

Abstract

The field of Software Engineering (SE) is the study of systematic and quantifiable approaches to software development, operation, and maintenance. This thesis presents a set of scalable and easily implemented techniques for quantifying and mitigating risks associated with the SE process. The thesis comprises six papers corresponding to SE knowledge areas such as software requirements, testing, and management. The techniques for risk management are drawn from stochastic modeling and operational research.

The first two papers relate to software testing and maintenance. The first paper describes and validates novel iterative-unfolding technique for filtering a set of execution traces relevant to a specific task. The second paper analyzes and validates the applicability of some entropy measures to the trace classification described in the previous paper. The techniques in these two papers can speed up problem determination of defects encountered by customers, leading to improved organizational response and thus increased customer satisfaction and to easing of resource constraints.

The third and fourth papers are applicable to maintenance, overall software quality and SE management. The third paper uses Extreme Value Theory and Queuing Theory tools to derive and validate metrics based on defect rediscovery data. The metrics can aid the allocation of resources to service and maintenance teams, highlight gaps in quality assurance processes, and help assess the risk of using a given software product. The fourth paper characterizes and validates a technique for automatic selection and prioritization of a minimal set of customers for profiling. The minimal set is obtained using Binary Integer Programming and prioritized using a greedy heuristic. Profiling the resulting customer set leads to enhanced comprehension of user behaviour, leading to improved test specifications and clearer quality assurance policies, hence reducing risks associated with unsatisfactory product quality.

The fifth and sixth papers pertain to software requirements. The fifth paper both models the relation between requirements and their underlying assumptions and measures the risk associated with failure of the assumptions using Boolean networks and stochastic modeling. The sixth paper models the risk associated with injection of requirements late in development cycle with the help of stochastic processes.

Keywords

software engineering, risk, execution trace, entropy, defect rediscovery, extreme value theory, Kappa distribution, G/M/k queue, binary integer programming, customer profiling, escalated requirement, assumption, stochastic modeling, Boolean network

Co-Authorship Statement

The following thesis contains material based on a previously published manuscripts co-authored with multiple colleagues. Andriy Miranskyy is the principal author of all material in this thesis. The details of the contribution of each author are given below. The names of authors are listed in alphabetical order per contribution type. Figures in brackets represent chapter numbers.

Matt Davison (2, 3, 4, 6), Nazim Madhavji (2, 6, 7), and Mark Reesor (3, 4, 6) facilitated modeling and interpretation of results. Mechelle Gittens (2) and Colin Taylor (2) collected data for the case study and helped validate practicality of the designed technique. Enzo Cialini (5), David Godwin (2, 5), and Mark Wilding (2) provided valuable feedback in validating the resulting models. Shariyar Murtaza (3) gathered data for the validation case study. Remo Ferrari (7), Shereen Ghobrial (7), Christine Giaraffa (7), and Quazi Rahman (7) contributed to analysis of the model's application.

Acknowledgments

It is a great pleasure to acknowledge the help and support that was given to me by my friends and colleagues.

First and foremost I would like to thank my supervisors and advisor Matt Davison, Nazim Madhavji, and Mark Reesor. I am sincerely thankful for all your time and energy that you have invested in me. Your assistance, remarkable expertise, and confidence in my abilities shaped me as a scientist.

I am extremely grateful to my coauthors and colleagues Enzo Cialini, Remo Ferrari, Shereen Ghobrial, Christine Giaraffa, Mechelle Gittens, David Godwin, Shariyar Murtaza, Quazi Rahman, Colin Taylor, and Mark Wilding for their contribution to papers and projects documented in this dissertation.

This research has been partially funded by the IBM Center for Advanced Studies. I am truly thankful to the outstanding staff of the Center for the help and support that they provided me over the years.

I would like to thank the entire Financial Mathematics and Software Engineering groups. Our friendly debates on seminars made priceless contributions to this thesis.

I would also like to show my appreciation to the administrative support provided by the magnificent staff of the Applied Mathematics department in particular and the University of Western Ontario for their constant assistants and support.

Of course, I would not be here without my family and friends; I owe my deepest gratitude to you.

Dictionary and Abbreviations

- **BIP** = Binary Integer Programming method.
- **Fat tailed (heavy-tailed) distribution** is a probability distribution having kurtosis > 3. A fat tailed random variable takes on extreme values more often than a normal distribution with the same mean and variance.
- **G/M/k** is a queue in which the inter-arrival time of requests are independent and identically distributed (iid) random variables from a general distribution, G, the service times are iid exponential random variables and k servers operate independently.
- **M/M/k** is a queue in which the inter-arrival time of requests are iid random variables from an exponential distribution, the service times are iid exponential random variables and k servers operate independently.
- **Program execution trace** is a sequential log of pertinent information captured during any particular run of software.
- **Software defect** is a fault in a computer program that produces an unexpected result or causes the program to behave in an unintended manner.

Table of Contents

CERTIFICATE OF EXAMINATION ... ii

Co-Authorship Statement ... v

Acknowledgments ... vi

Dictionary and Abbreviations ... vii

Table of Contents ... viii

List of Tables ... xiii

List of Figures ... xv

Chapter 1 ... 1

1 Introduction ... 1

1.1 Outline ... 5

References ... 8

Chapter 2 ... 10

2 SIFT: A Scalable Iterative-Unfolding Technique for Filtering Execution Traces ... 10

2.1 Introduction ... 10

2.2 Related Work ... 14

2.3 Method Description ... 17

2.3.1 The Iterative-Unfolding Approach ... 17

2.3.2 Algorithms ... 18

2.4 Analysis ... 28

2.4.1 Efficiency ... 28

2.4.2 Method Accuracy ... 32

2.4.3 Iteration-unfolding overheads ... 33

2.5 Implementation 34

2.6 Validation Case Study.. 35

2.7 Conclusion And Future Work 42

References 43

Chapter 3 46

3 Using Entropy Measures for Comparison of Software Traces 46

3.1 Introduction.. 46

3.2 Entropies and Traces: definitions... 50

3.2.1 Extraction of probability of events from traces 50

3.2.2 Entropies and traces 51

3.3 Usage of entropies for classification of traces 52

3.3.1 Measure of distance between a pair of traces 54

3.3.2 Trace-ranking algorithm 55

3.3.3 Traces ranking algorithm: efficiency 57

3.3.4 Entropies as fingerprints: drawback.. 58

3.4 Validation case study 58

3.4.1 Analysis of individual entropies 61

3.4.2 Analysis of the complete set of entropies 68

3.5 Summary 69

References 70

3.6 Appendix: Approximation of Equation (3.8).. 72

Chapter 4 74

4 Metrics of Risk Associated with Defects Rediscovery 74

4.1 Introduction.. 74

4.2 Related Research 76
4.3 Metrics of Risk 77
4.3.1 Metrics Application 77
4.3.2 Formulation of Metrics 80
4.4 Case Study 86
4.4.1 Finding a Suitable Distribution 89
4.4.2 Application of the Metrics 95
4.4.3 Threats to Validity 102
4.5 Conclusions 103
References 103
Chapter 5 105
5 Selection of Customers for Operational and Usage Profiling 105
5.1 Introduction 105
5.2 Related Work 108
5.3 Qualitative Analysis Of Customers 108
5.4 CUSTOMER SELECTION TECHNIQUE 110
5.4.1 Minimization of Customer Set 110
5.4.2 Prioritization of Customers within the Minimal Set 113
5.5 Validation Case Study 114
5.5.1 Exploratory Analysis 114
5.5.2 Selection of the Minimal Set of Customers 115
5.6 Summary 118
References 118
Chapter 6 120

6 Modelling Assumptions and Requirements in the Context of Project Risk 120
6.1 Introduction 120
6.2 Related work 123
6.3 Requirements & Assumptions 125
6.3.1 Assumptions Formalization 125
6.3.2 Requirements Formalization 127
6.3.3 Requirements & Assumptions Interaction 128
6.4 Modelling tools 129
6.4.1 Boolean network 130
6.4.2 Modelling Event Arrival 131
6.5 Predicting risk at time *t* 135
6.5.1 Risk metrics 135
6.5.2 Single-run Algorithm: System State at Final Time 137
6.5.3 Multiple-runs Algorithm: System State at Final Time 139
6.6 Simulation Example 140
6.7 Conclusions & Future Work 146
References 146
Chapter 7 149
7 Managing the Escalation of Requirements 149
7.1 Introduction 149
7.2 Modeling the Escalation of Requirements 151
7.2.1 Modeling the Escalation of Known Requirements 152
7.3 Conclusions and Future Work 160
References 160

Chapter 8 161
8 Conclusions and Future Work 161
References 163
Curriculum Vitae 165

List of Tables

Table 1. Papers: summary information 6

Table 2. Applicability of the papers to software development phases (X marks applicable area) 6

Table 3. Descriptive statistics of traces 36

Table 4. Dictionaries of a trace given in Figure 11 51

Table 5. Example: Relation between traces and defects. 56

Table 6. Example: Traces sorted by distance and ranked 57

Table 7. Example: Top 1-4 defects 57

Table 8. Descriptive statistics of length of traces 60

Table 9. Fraction of correctly classified traces in Top X for 1) $H_E\left[\alpha(t;l,c);q\right]$ with $E \in (L,R,T)$, $q \in (10^{-5},10^{-4})$, $l=3$, and $c=FDT$, and 2) set of entropies Λ; based on 10-fold cross validation. Average fraction of correctly classified traces in 10 folds is denoted by “Avg.”; plus-minus 95% confidence interval denoted by “95% CI”. 65

Table 10. Percent of correctly classified traces in Top X for $H_E\left[\alpha(t;l,c);q\right]$, $E=L$, $l=3$, and $q=\{10^{-4},10^{-5}\}$ 67

Table 11. AIC 90

Table 12. Values of variables 93

Table 13. Results of the G/M/k model for v.4, second year. 102

Table 14. Customer prioritization criteria 110

Table 15: Example. Defects' discovery 113

Table 16. Percentage of the total number of customers needed to cover *X*% of defects discovered at least *Y* times 117

Table 17. Example 4.1. State changes of assumptions. 132

Table 18. Assumptions properties........ 142

Table 19. Requirements properties 142

Table 20. Metrics values at $T_f = 1$ (± denotes standard deviation) 145

Table 21. Setup parameters........ 155

Table 22. Expected escalation time 157

List of Figures

Figure 1. An example of a trace 11

Figure 2. Algorithm for comparing two processes 21

Figure 3. Comparison of two uncompressed processes. Upper dashes depict functionality present only in Process 1; lower dashes – functionality present only in Process 2; no dashes – common functionality. 22

Figure 4. Algorithm for measuring distance between traces. 25

Figure 5. Algorithm for comparing a single trace *t* against a set of traces *S*. 27

Figure 6. Algorithm for comparing traces within a given set *S*. 31

Figure 7. Timing of comparing a trace against a set of traces; timing for each draw is denoted by circles; values are plotted on the left axis. Solid line shows the number of comparisons in the lower pattern; dotted line in the upper pattern; values of both lines are plotted on the right vertical axis 38

Figure 8. Timing of comparing traces within a given set 39

Figure 9. Number of traces remaining after each iteration of comparing traces within a given set 39

Figure 10. Predicted time (dotted lines represent 95% confidence bands) for (a) $P_1(t,S)$ – linear, and (b) $P_2(S)$ – quadratic, based on extrapolation of the fitted regression. 41

Figure 11. An example of a trace 46

Figure 12. Distribution of the number of traces per defect (version) 60

Figure 13. Dictionary size for various values of l and c 61

Figure 14. Interpolated average fractions of correctly classified traces in Top 5 (based on 10-fold cross validation) for $E = L$ and $c = FDT$. for different values of l and q 63

Figure 15. Fraction of correctly classified traces in Top 5 for $E = L$, $l = 3$, $q = 10^{-5}$, and $c = FDT$. Solid line shows the average fraction of correctly classified traces in 10 folds; dotted line shows pointwise 95% confidence interval (95% CI) of the average. 64

Figure 16. Average fraction of correctly classified traces in Top 5 for various values of l; $E = L$, $q \in (0, 10^{-5}, 10^{0}, 10^{2})$, $c = FDT$ 66

Figure 17. Average fraction of correctly classified traces in Top 5 for various values of q; $E = L$, $l \in (1,3,7)$, $c = FDT$ 68

Figure 18. $N(t)$: total number of defects discovered up to time t. 87

Figure 19. $R(0,t)$: total number of rediscoveries up to time t. 88

Figure 20. L-moments ratio diagram of D_i for all releases per year (years 1 – 5). The hollow circles denote each of the yearly datasets of D_i. The diagram shows the fits of the following distributions: Exponential (EXP), Normal (NOR), Gamma (GUM), Rayleigh (RAY), Uniform (UNI), Generalized Extreme Value (GEV), Generalized Logistic (GLO), Generalized Normal (GNO), Generalized Pareto (GPA), generalization of the Power Law, Pearson Type III (PE3), and Kappa (KAP). Kappa distribution applicability space is a plane bounded by GLO distribution line above and the "Theoretical limits" line below and is not shown on the legend. Based on this figure, Kappa distirbution is the only one that is applicable to modeling each of the datasets. 90

Figure 21. QQ plot of the empirical vs. PE3 distributions' quantiles. 91

Figure 22. QQ plot of the empirical vs. KAP distributions' quantiles. 91

Figure 23. QQ plot of the empirical vs. Compound distributions' quantiles.................... 94

Figure 24. Plot of the empirical cdf vs. Compound Kappa distribution theoretic cdf...... 95

Figure 25. M_1: expected number of defects rediscovered more than d times during the 2nd year after GA date. 96

Figure 26. M_3: expected total number of rediscoveries for defects with number of rediscoveries above d during the 2nd year after GA date. 98

Figure 27. M_5: probability that the total number of rediscoveries will not exceed L during the 2nd year after GA date. 99

Figure 28. Estimate that the total number of rediscoveries will not exceed M_6 with confidence level α. 100

Figure 29. Density of requests inter-arrival times for v.4, second year......................... 101

Figure 30. Total number of discovered defects vs. average number of rediscoveries per customer. Dotted lines depict borders of quadrants described in Table 14. 115

Figure 31. Percentage of the total number of customers needed to cover a certain percentage of defects of interest. 116

Figure 32. Percentage of the total number of customers needed to cover a certain percentage of defects of interest (log-scale). 117

Figure 33. Example 4.1. Set up of assumptions for a. Configuration I; b. Configuration II. Solid arrows denote standard relationship, dotted arrows denote key relationship........ 131

Figure 34. Example 4.2. Five random realizations of $I(r,t)$ 134

Figure 35. Simulation setup. Circles denote assumptions, squares denote requirements. Solid arrows denote standard relationship, dotted arrows denote key relationship........ 141

Figure 36. The value of $\hat{V}(\cdot,t)$ 143

Figure 37. The value of $\hat{C}(\cdot,t)$ 144

Figure 38. The value of $\hat{U}(\cdot,t)$ 144

Figure 39. The value of $\hat{I}(\cdot,t)$ 145

Figure 40. Dependencies among requirements 155

Figure 41. Case 1. Expected priority 156

Figure 42. Case 2. Expected priority 157

Figure 43. Case 3. Expected priority 158

Chapter 1

1 Introduction

In this section, we give a brief exposition to the field of software engineering and the challenges faced in the field. The term "Software Engineering", and the engineering subfield it describes, was coined in 1968 at the NATO Software Engineering Conference [1]. The IEEE Computer Society Software Engineering Body of Knowledge defines software engineering as:

> "(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)" [2].

This discipline was created to address the "software crisis" increasingly apparent [1], [3] at the time. This crisis described the difficulty of writing correct and maintainable programs as computational power and the concomitant complexity of problems that can be tackled increased. The crisis manifested itself both in: (i) unmanageable projects running over budget and late[1] and (ii) low quality, inefficient software not meeting original requirements. In some cases projects failed completely, being unable to deliver a final product. In order to tackle these issues, well defined and structured approaches had to be developed.

The complexity problem and the need for defined processes can be described using analogies from building construction. While most people can hammer a nail into a wood board, a much smaller fraction of the population (your humble author excluded) is capable of building a doghouse with an even smaller number of people being capable of building a wooden cabin. Increase in the size and complexity of a project demands not

---

[1] Based on industrial surveys and statistical data, even in the modern era, the average project is 6 to 12 month behind schedule and 50 to 100 percent over budget [4].

only increased craft skills but also the ability to plan, design, build and test the final product.

Similar to other engineering disciplines, software engineering is divided into a number of knowledge areas [2], [5]. We now describe a few such areas relevant to this thesis. The first four areas can be mapped to specific development phases:

1. Software requirements: deals with elicitation, analysis, specification, and validation of requirements for a given software project.

2. Software design: generates high-level designs (also called architecture) depicting the components and their interfaces, based on specifications elicited during the software requirements phase. Once the high-level design is complete, low-level design of specific components can be created.

3. Software construction: relates to the actual implementation (coding and unit testing) of the product based on design specifications.

4. Software testing: verifies that the implementation satisfies the specifications and is free of defects. Once the testing is complete, the product is deployed in the field.

In practice, the development is done using an iterative and incremental approach [6]: an organization will pass through multiple iterations of requirements, design, construction, and testing between the initial planning and final deployment of the software product.

5. Software maintenance: provides support by answering questions and fixing defects that have escaped the testing team and were identified by customers in the field (after product delivery). In addition, software maintenance may also deal with changes to product functionality (also called software evolution) satisfying additional customer requirements arising during product exploitation use.

The previous five knowledge areas map to specific phases of software development. The remaining two are more general.

6. Software engineering management: relates to project management (and measurement) of software engineering.

7. Software quality control: corresponds to the qualities of the intended system (e.g., reliability of the system, performance, usability, interoperability, portability, maintainability, and others). This area is tightly related to all of the areas listed above.

A significant amount of work has been done improving the software development process and integrating these changes into the industry. However, an analysis (based on literature and empirical evidence) published in 2003 concludes:

> "In a discussion of software engineering and society in 1968 [1], Kolence suggests that 'the basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time.' Empirical evidence of software engineering projects suggests this crucial issue remains valid 35 years after it was stated. It appears that as fast as software engineering makes progress, so the demands made on it continue to increase beyond its capabilities" [7].

What is the current state of the process maturity in the industry? An approach for process improvement, called Capability Maturity Model Integration (CMMI) [8], was developed by the Software Engineering Institute of Carnegie Mellon University (SEI). The CMMI helps "integrate traditionally separate organizational functions, set process improvement goals and priorities, provide guidance for quality processes, and provide a point of reference for appraising current processes." [8] The CMMI defines five levels of process maturity [8]: from the lowest (CMMI-1) corresponding to chaotic, poorly controlled, and reactive processes; to the highest (CMMI-5) corresponding to proactive, well defined processes with focus on constant improvement.

The SEI conducts regular surveys to determine the maturity distribution of processes in the industry. Based on the survey of process maturity profiles [9] the number of organizations with chaotic processes or defined reactive processes (CMMI -1 and -2) went down from ≈35% in 2002 to ≈28% in 2010. The number of organizations with proactive processes (CMMI-3) went up from ≈33% in 2002 to ≈55% in 2010. However, the number of organizations with highest levels of process maturity (CMMI -4 and -5) focusing on well established, measured and controlled processes and with continuous process improvement went down from ≈23% to ≈10%.

Why is the chaos in Software Engineering higher than in other engineering disciplines [10]? Why is the fraction of organizations with highest process maturity decreasing [9]? Increasing complexity is one of the major contributing factors to the problem [7]. However, there exist additional economic and legal reasons.

The problem is two-fold:

1. Many of the techniques created to improve software processes are non-scalable [11] – as projects get larger time and resource constraints force some processes[2] to be sacrificed at least part of the time.

2. Even if a technique is applicable to a given project, it may not be enforced by the organization due to corporate culture [10]: employees' performance evaluation may not take into account the use of proper processes.

Potential solutions to this problem are complicated by the fact that the majority of software products (unlike products created using other engineering disciplines) include "as-is" clauses in their licenses stating that a given software vendor does not provide any warranty and shall not be deemed liable for any damage caused by its software products.

---

[2] An example of such process is software inspection (structured peer-reviewed process aimed at finding defects in development documents, such as programming code and design) [12].

This can lead to the situation where the final quality of the product is considered non-critical by some developers. This decreases the economic incentive for software developers to properly engineer solutions.

How should the situation be improved? The first aspect of the problem can be solved by developing fast and scalable solutions supported by proper empirical studies [13], [14]. The second aspect does not have a straightforward solution: it is extremely difficult to change corporate culture [15]. In order to address this issue, newly developed techniques and processes should be capable of being integrated into existing processes without draining significant resources. Namely, they should satisfy the following requirements: a) be easy to implement and b) be easy to automate. Once implemented, a process should run automatically and deliver regular reports in a comprehensible format.

This thesis describes a set of models, techniques, and metrics (satisfying the above mentioned requirements) that can help in the analysis of computer software during various phases of software development.

1.1 Outline

Many problems in the Software Engineering domain are similar, from the mathematical perspective, to problems in other disciplines, such as Financial Engineering. Therefore, similar tools may be applied to Software Engineering and Financial Engineering problems. One example is the usage of stochastic tools for modeling stock prices and requirement attributes. Another example is the usage of Extreme Value Theory for modeling the probability of rare events, such as high magnitude earthquakes and software defects with a high number of rediscoveries. The thesis consists of six papers (one paper per chapter) that use mathematical tools (from the domains summarized in Table 1) to develop a set of techniques to assess and mitigate risks associated with particular phases of software development (given in Table 2).

Table 1. Papers: summary information

Chapter #	Topic	Publication type	Tools	Reusing tools from
2	Technique for selection of software traces	Conference proceedings (best student paper)	Heuristics	Algorithms
3	Technique for selection of software traces using entropies	Submitted to Information Sciences	Entropies	Information Theory
4	Metrics quantifying risk associated with defect rediscovery	Manuscript	Extreme value theory, heavy-tailed distributions	Risk Management and Operational Research
5	Model for selection of a minimal set of customers for profiling	Workshop proceedings	Binary integer programming	Operational Research
6	Model of relations between requirements and underlying assumptions	Conference proceedings (short version)	Stochastic models, Boolean Networks	Stochastic Modeling
7	Models for managing injection of requirements late in development cycle	Workshop proceedings	Stochastic models	Stochastic Modeling

Table 2. Applicability of the papers to software development phases (X marks applicable area)

Chapter #	Development Phases					Overall product quality	Project management
	Requir-ements	Archi-tecture	Coding	Testing	Mainte-nance		
2				X	X		
3				X	X		
4					X	X	X
5					X	X	X
6	X						
7	X						

The first and second papers address issues arising in the testing and maintenance phases of software development. The analysis of execution paths (also known as software traces) collected from a given software product can help in software testing and software maintenance. Unfortunately, techniques operating on traces containing full execution details are resource-intensive. In the first paper, given in Chapter 2, we propose a “fingerprint”-based iterative-unfolding technique for prompt selection of a subset of traces relevant to a given task. Once the subset is selected, it can be passed to external tools for further analysis. In Chapter 3, we study the applicability of extended entropies in

the role of fingerprints. The techniques in these two papers can accelerate problem determination of defects discovered by customers, leading to improved organizational response (thus increasing customer satisfaction) and to easing of resource constraints.

The third and fourth papers, presented in Chapters 4 and 5, help in addressing issues related to software maintenance and overall quality. Techniques presented in these papers also help project managers in resource allocation.

The third paper analyzes rediscovery of defects by customers and establishes a set of metrics (based on risk management and operational research apparatus) needed to quantify the risks associated with defect rediscovery. The metrics are designed to help: the QA team to assess their processes, the support and maintenance teams to allocate their resources, and the customers to assess the risk associated with the use of the software product.

Collecting information about product usage by customers (operation profiling) helps testers to build realistic workloads, covering functionality executed by customers, hence reducing risks associated with inadequate product quality. However, it is impossible to gather such information from all customers due to resource and legal constraints. The fourth paper establishes a relation between defects encountered by customers and their operational profiles. We then build a model for selecting a minimal set of customers that should be profiled to gather information about users' behaviour.

The fifth and sixth papers deal with problems from the domain of requirements engineering.

Each requirement collected during requirements elicitation has an underlying assumption. However, incorrect assumptions can lead to software problems. In order to quantify the risks associated with such problems a stochastic model of relations between requirements and underlying assumptions is established in the fifth paper (Chapter 6).

The sixth paper, presented in Chapter 7, defines stochastic models for managing risk associated with requirements injected late in the software development cycle (we call this event escalation). The models can help in predicting escalations of the existing requirements and in allocating resources to handle the arrival of unknown requirements. Finally, Chapter 8 concludes the thesis.

References

[1] P. Naur and B. Randell, "Software Engineering: Report of a conference sponsored by the NATO Science Committee Garmisch Germany 7th-11th October 1968," Scientific Affairs Division, NATO, 01-Jan-1969.

[2] A. Abran, P. Bourque, R. Dupuis, J. Moore, and L. Tripp, *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2004.

[3] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, pp. 859–866, Oct. 1972.

[4] E. Yourdon, *Death March*, 2nd ed. Prentice Hall, 2004.

[5] "Industry implementation of International Standard ISO/IEC 12207: 1995. (ISO/IEC 12207) standard for information technology - software life cycle processes - life cycle data," *IEEE/EIA 12207.1-1997*, 1998.

[6] C. Larman and V. R. Basili, "Iterative and incremental developments. a brief history," *Computer*, vol. 36, no. 6, pp. 47 - 56, Jun. 2003.

[7] C. L. Simons, I. C. Parmee, and P. D. Coward, "35 years on: to what extent has software engineering design achieved its goals?," *Software, IEE Proceedings -*, vol. 150, no. 6, pp. 337 - 350, Dec. 2003.

[8] CMMI Product Team, *CMMI for Development, Version 1.3; CMU/SEI-2010-TR-033*. Software Engineering Institute of Carnegie Mellon University, 2010.

[9] CMMI Appraisal Program Team, "Process Maturity Profile. CMMI For Development SCAMPI Class A Appraisal Results 2010 Mid-Year Update," Sep-2010.

[10] D. L. Parnas, "Risks of undisciplined development," *Commun. ACM*, vol. 53, pp. 25–27, Oct. 2010.

[11] A. V. Miranskyy et al., "SIFT: a scalable iterative-unfolding technique for filtering execution traces," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pp. 274-288, 2008.

[12] M. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, vol. 12, no. 7, pp. 744-751, 1986.

[13] E. J. Weyuker, “Software engineering research: from cradle to grave,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pp. 305–311, 2007.

[14] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Softw. Engg.*, vol. 14, pp. 131–164, Apr. 2009.

[15] J. P. Kotter and J. L. Heskett, *Corporate Culture and Performance*. Free Press, 1992.

Chapter 2

2 SIFT: A Scalable Iterative-Unfolding Technique for Filtering Execution Traces

Comparing program execution traces can be useful for numerous purposes, such as software testing, system security analysis, program comprehension, software evolution and other areas of software development. Unfortunately, trace comparison techniques that operate on execution traces containing full execution details are too slow for use in large-scale production system environments. In order to speed up the comparisons, we propose a technique (called SIFT) for "filtering-out" irrelevant traces from a given set so that only the relevant few residual traces are then used for comparison. Our solution involves multiple levels of trace compression, each with a different degree of abstraction. These traces are compared iteratively while filtering out dissimilar traces. This chapter describes the compression and comparison algorithms. Prototype results from a significant case study show that the SIFT approach is efficient and scalable for use in an industrial software development environment.

2.1 Introduction

The comparison of program execution traces (see an example trace in Figure 1) is important for a number of problem areas in software development and use. In the area of testing, for example, it can be used to: (1) determine how well user execution paths (traces collected in the field) are covered in testing [4, 7, 25]; (2) detect anomalous behaviour arising during a component's upgrade or reuse [21]; (3) map and classify defects [13, 27, 31]; (4) determine redundant test cases executed by one or more test teams [30]; and to (5) prioritize test cases (to maximize execution path coverage with a minimum number of test cases) [8, 23]. Trace comparisons are also used in operational profiling (for instance, mapping frequency of use of execution paths by different classes of users.) [25] and intrusion analysis (e.g., deviations of field execution paths from expectations) [17].

For some problems, such as test case prioritization, traces gathered in a condensed form (such as a vector of executed function names or caller-callee pairs) are adequate [8]. However, for others, such as the detection of missing coverage and anomalous behaviour using state machines, detailed execution paths are necessary [4, 21]. Also, in some situations, the time required for analysing traces can be extremely important, for example when a customer support analyst is using traces to map a reported defect onto an existing set of defects, or when a development analyst is working with the testing team to identify missing coverage that resulted in a field defect.

```
process_id = 133 thread_id = 15 node = 0
1           f1 entry
2           | f2 entry
3           | | f3 entry
4           | | f3 data [probe 1]
5           | | f3 data [probe 2]
6           | | f3 exit
7           | f2 exit
8           f1 exit
```

In general, a trace can be thought of as a sequential log of pertinent information captured during any particular execution-run of software. This trace shows program flow entering functions[3] *f1, f2, and f3, and eventually exiting these functions and, while in function f3, it reached specific data points (probe 1 and probe 2), which were manually set by the developer as points of interest.*

Figure 1. An example of a trace

Research Problem and Practical Motivation: Unfortunately, comparison techniques that use full execution paths (i.e., do not use some abstracted versions of the paths) lack speed, which can become critical when using numerous large execution traces. For

[3] In this case a function is equivalent to a subroutine.

example, the kTail-based algorithms[4] applied to traces from a reference system, called Object Flattener, (for the purpose of creating finite state automata – which would involve comparing given traces) did not terminate after 24 hours of execution [4]. While the kBehavior algorithm [4] accomplished this task in “minutes”, their paper does not mention the size or the number of traces involved in Object Flattener. Subsequently, we applied kBehavior prototype tool [15] on traces from our environment. With only 2 traces totalling 426 elements (the smallest in our set of traces), the tool took 9 minutes; with 36 traces totalling 8625 elements (considered small in our environment), it did not terminate after 36 hours. Finally, various permutations of up to 57 traces and up to 68,705 elements consistently caused the kBehavior prototype to crash. These experiences highlight the need for speed and robustness of the solutions.

In our development environment, we are faced with a distributed, multi-process, and multi-threaded system of over 10 million lines of uncommented source code developed over 15 years, with over 100 thousand traces (many with millions of elements per trace) from the testing phase. There are hundreds of thousands of installations worldwide of the system, in different configurations. As a result, a critical issue surfaced as to how quickly the testing organization could identify those test cases that match the traces collected from the field upon recognition of a defect or a problem such as a logic error. While there are many different approaches to performing trace comparisons, they are not considered workable, as described earlier, for the large and complex system we are dealing with. The described need to match field traces with test cases quickly, together with a lack of reliable and scalable tools for doing this, motivated us to investigate alternate solutions.

Solution Approach: To speed up comparison of traces, we propose that traces first be filtered out from the given set, rejecting those that are not going to match with the test cases and then only the remaining few be compared for target purposes. The underlying

---

[4] The algorithms generate Finite State Automata models.

assumption is that most traces in a given set will not be the same; few will be similar, and even fewer will be identical. We validated this assumption in our sample set of traces by manual inspection of the dataset. Thus, filtering out irrelevant traces is a key to speeding up the comparison process.

This strategy is implemented in our solution called the **S**calable **I**terative-un**F**olding **T**echnique (SIFT). Basically, the collected traces are first compressed into several levels prior to comparing them. Each level of compression uses a unique signature, which we call a "fingerprint"[5]. Then, starting with the highest level of compression the traces are compared, and unmatched ones rejected, while iterating through the lower levels until the comparison process is complete, leaving only traces that match at the lowest (or uncompressed) level. The SIFT objective ends here. The matched traces can then be passed on to external tools for further analysis such as code coverage, security breaches, and operational profiling.

Case study: The prototype results from a case study we conducted show that the approach is scalable for use in large-scale software system development environments. That is, in no more than four iterations (depending on the context of the specific problem being solved), dissimilar traces are rejected, leaving only the residual similar traces. The case study was conducted on 1416 multithreaded test case traces collected from the system under study (SUS), with an average length of 1.93×10^6 elements (maximum 1.55×10^8 elements) per trace. The first test was comparing a trace against a set of 1,000 traces (e.g., useful for coverage analysis); the average time was 4 seconds. The second test was "within-set" filtering and clustering (e.g., useful for periodic profiling of usage similarity among a class of users or a class of test cases); the average of comparing multiple times within a set of 1,000 traces was only 44 minutes.

---

[5] The fingerprint of the next iteration always contains more information than the fingerprint of the previous iteration, hence the term *unfolding*.

Based on the extrapolation of the timing data obtained from the case study, it should take 39 seconds on average to filter out 10,000 different traces, leaving, as a residue, a few uncompressed traces that are within the user-defined threshold of similarity. Similarly, filtering (and clustering) within a set of 10,000 traces should take 2.6 days on average. We consider the "within-set" performance as quite reasonable, and encouraging, given that such profiling would occur only periodically in a product's life. Considering the lack of readily available solutions for use in industrial-scale environments, the proposed approach is both elegant and effective.

The rest of the paper is structured as follows: Section 2.2 reviews related literature. Section 2.3 details the SIFT and associated algorithms. Section 2.4 analyses the efficiency and accuracy of the SIFT. Section 2.5 describes an implementation and use of the proposed approach. Section 2.6 describes the case study. Finally, Section 2.7 gives conclusions and describes future work.

2.2 Related Work

A variety of different approaches (such as, automata, signals, call-graphs, compression without information loss, and clustering) exist in dealing with execution traces for various target purposes such as finite automata representation, visualization, test-case prioritisation, and failure classification. These are overviewed below.

Finite State Automata representation: There is a family of kTail-based algorithms that can be used to find differences in execution path coverage between traces [2, 3, 20, 28]. These algorithms generate Finite State Automata (FSA) interaction models by combining traces into prefix tree automaton. Observed behaviour is generalized by merging states that cannot be distinguished from the outgoing paths of length k (merge shared k-future states). The original kTail algorithm was proposed by Biermann and Feldman [2]. Cook and Wolf [3] introduce an additional reduction step to the kTail approach. Reis and Renieris' [28] algorithm reduces the FSA if two states share at least one k-feature. Finally, Mariani and Pezze [20] introduced the kInclusion extension to kTail (two states

are merged if a k-feature of the first state is included in the k-feature of the second state). All of the algorithms above need to process all traces first before generation of the interaction model can begin. Mariani and Pezze [21] developed an algorithm, called kBehavior, that overcomes this issue and works incrementally. These techniques may be used to determine how well user execution paths (traces collected in the field) are covered in testing [4], detecting anomalous behaviour arising during a component's upgrade or reuse [21], and general program comprehension [28].

Signal representation: Kuhn and Greevy [16] visualize multiple execution traces in signal form, discarding information about function names. Once conversion to signal form is complete, a Dynamic Time Warping (DTW) pattern recognition technique is used to find similar patterns between traces. This approach is used to compare detailed execution traces for program comprehension. Unfortunately, patterns of execution paths containing different function calls may have similar shapes. This is especially true for large software systems consisting of multiple processes. Therefore, it is important to group (pre-filter) the traces containing similar execution paths, before conversion to the signal form. Once pre-filtering is complete, an analyst can apply the DTW technique to each of the groups separately.

Call graph representation: Another approach patented by Avvari et al. [1] is the idea of creating and analysing call graphs. Execution paths (EP) are first converted into call graphs and then compared in graph form. In practice, the quantity and complexity of EP in a large software system would not allow one to perform this comparison directly in a feasible amount of time. For example, for a large software system, the number of test cases can be of order of 10^5, with the number of records per EP of the order of 10^6. The performance of the publicized algorithms is, at best, of $\mathcal{O}\left[\sqrt{|V|}|E|\log\left(|V^2|/|E|\right)/\log(|V|)\right]$, where V is the number of vertices and E is the number of edges in a given EP call graph (see [9, 10] for details).

Lossless compression techniques: With these techniques, it is possible to reconstruct the exact original data from the compressed data. Renieries et al. [29] introduce lossless compression techniques for source-code-level traces that lead to significant compression of the original traces. Techniques, such as that designed by Hamou-Lhadj and Lethbridge [12] can be used for visualizing traces in a compact form, which can be useful for viewing several traces on a display screen during software maintenance.

Lossy compression techniques: With these techniques, reconstruction of the exact original data from the compressed data is not possible. There are both short and long execution sequences, used for various purposes. In the realm of short sequences, for example, Elbaum et al. [8] found that function-name-level execution traces can be useful for test case prioritization. Rothermel et al. [30] and Masri [22] used the same type of traces to perform test suite reduction/minimization by identifying redundant traces. Greevy et al. [11] explore relations between features (function names) extracted from traces and software entities for software evolution analysis. Yuan et al. [31] found that for system call defects, caller-callee-pairs-level execution traces (with parameter information) were effective for mapping a new problem to an existing one. In the realm of long sequences, for example, Miranskyy et al. [23] show that sequences of length 3 or more are potentially important for test case prioritization. Elbaum et al. [7] found that sequences of length 5 are useful for fault detection. Dalmeier et al. [5] studied sequences of various lengths (no more than 8) to localize defects. Lee et al. [17] found execution sequences of length 7 and 11 to be useful for intrusion detection.

Trace clustering techniques: In addition, researchers have created techniques to compare other types of information. For example, a number of studies exist on clustering execution profiles collected from software users. These profiles are not focused on execution traces and can include additional information about a particular software system, such as covered code-blocks, heap size, CPU load, etc. For example, Podgursky et al. [27] and Haran et al. [13] use various techniques to cluster such profiles. These clusters can be used to classify a software system’s failures. However, they do not

address the challenge of comparing detailed uncompressed traces so the need to filter the traces remains.

While there are many techniques, as described above, researchers have not considered the idea of filtering-out traces to improve comparison of uncompressed traces. This is the bounded scope that is addressed by our work described in the next section.

2.3 Method Description

In this section, we first describe the basics of the SIFT approach in Section 2.3.1. As overviewed in the introduction section, this approach filters out traces from a given set that are not going to match with the test cases, leaving a few for detailed comparison. Section 2.3.2 then describes the algorithms underlying this approach. The analysis of these algorithms is carried out in Section 2.4.

2.3.1 The Iterative-Unfolding Approach

Further to the introductory description earlier, the idea behind the SIFT approach is two-fold.

Unfolding: First, traces to be compared can exist at different levels of compression. For example, at the lowest level of compression, a trace would be at the level of detail captured from program execution, where a sequence of function calls is represented as a string of calls. A slightly higher level of compression could be, for example, where this string of functions calls is broken down into “caller-callee” pairs. A yet higher level of compression could be just a list of function names. The type of compression applied and the number of compression levels is analyst defined and should be selected in such a way that the current iteration compression technique retains less information than the next iteration.

Also, applying compression leads to loss of information in the resultant compressed trace. The type of information lost depends on the compression technique applied. Furthermore, to obtain the various higher-level traces, different compression techniques are applied

directly to the program-level (or lowest-level) trace. Compression techniques and compression levels are independent of each other. Note that any compression technique can be used for trace compaction, as long as a certain measure of distance can be calculated between a pair of compressed traces.

The full scope of the type of data involved in a trace can include a wide variety of program elements such as: events, logic-based points in the program flow, store & retrieve transaction points, and process enaction or termination points. We define a *process* as an instance of a sequentially executed computer program.

Iterative: The second idea behind the SIFT approach is that it proceeds by comparing stored traces at the highest level of compression and, based on the outcome of this comparison (that is, a set of matched and unmatched traces), discards the unmatched set of traces and proceeds to compare the matched set of traces at the next lower level of compression until a terminating condition is satisfied. The terminating conditions are: (1) the number of traces remaining for comparison is below a certain threshold; (2) no lower levels of compression of traces exist; and (3) practical conditions such as exhausting time and resources.

A benefit of this approach is that it makes comparison of large program traces or large volumes of traces practical. Later, we discuss the various permutations of trace comparison situations and how our approach fares with these.

2.3.2 Algorithms

It is important to understand the different situations where trace comparisons can be useful, for example, how test cases relate to each other and to field execution of the software system. In Section 2.3.2.1, we discuss various situations of interest for trace filtering. Two core comparison algorithms are discussed in Section 2.3.2.2. Recall that the comparisons proceed iteratively, from higher to lower levels of compression; the algorithms are further described in Section 2.3.2.3.

There are several fundamental types of situations for comparing execution traces:

(a) A single trace *t* against a set of traces[6] *S*. We can represent this comparison process by a function P_1 that takes as input two variables of interest, trace *t* and a set of traces *S*, and outputs a subset of traces S_1 closest[7] to *t*:

$$P_1(t, S) \rightarrow S_1.$$

One example of this situation is where the single trace is captured from the system's use in the field, hereafter called User Trace (UT); by contrast, the set represents the traces captured from the execution of test cases, hereafter called House Traces (HT). This is useful for identifying a subset of HTs that match the given UT for a purpose such as coverage analysis, or for that matter, for identifying mismatches for the purpose of proactively creating new test cases.

(b) Within a given set of traces *S*. Here, the comparison function P_2 clusters *S* into *L* subsets of similar traces:

$$P_2(S) \rightarrow \{S_1, S_2, \ldots, S_L\}.$$

One example of this situation is comparing a set of UTs against itself to profile subsets of customers with similar system usage needs. The outputs of the comparison process are subsets of similar and different traces.

In the next section, we describe how the comparison of two given traces is performed.

---

[6] This approach can be further generalized to comparing two sets of traces, see [24] for further details.

[7] See Section 4.3.2.1.2 for a complete discussion of our distance metric.

2.3.2.1 Core Algorithms: Fingerprints, Processes, and Traces

There are a number of algorithms that are used for the SIFT approach. For the sake of clarity, we will first explain each algorithm and then continue with the description of the overall approach.

First, as described in Section 2.1, there is the notion of a "fingerprint". The fingerprinting technique is described in Section 2.3.2.1.1. Also, we introduced the notion of a process in Section 2.3.1. Traces contain information about multiple processes executed in parallel (mostly independently). To align execution sequences pairwise comparison of processes has to be implemented. The algorithm for comparing a pair of processes is described in Section 2.3.2.1.2. We then describe the algorithm for comparing a pair of traces, building upon process comparison, in Section 2.3.2.1.3. The overall approach binding fingerprints, processes and traces is described in Section 2.3.2.1.4.

2.3.2.1.1 Fingerprints

A "fingerprint" of a process describes the uniqueness of the process in terms of the call sequence, elements of contextual information, and other relevant information that make up the fingerprint. The first technique for creating fingerprints would be the collection of component names along with the frequency of occurrence contained in each process. On average, the number of components per process is of order 10^1. In most cases, it should not be bigger than 10^2. Similarly, information about function names can be collected.

In order to collect the next set of fingerprints we use a concept of *l*-words (also known as N-grams [31]) to represent execution sequences. An *l*-word represents a continuous substring of length *l* from a string. We then collect information about all possible entry, exit, and probe points (defined manually by software developers at important places inside the functions) and their frequency for each process (1-words) and end up with all possible pairs of entry, exit, and probe points (2-words). For example, for the process given in Figure 1, the set of all possible 1-words will be given by the set $\left\{f_1^+, f_2^+, f_3^+, f_3^1, f_3^2, f_1^-, f_2^-, f_3^-\right\}$, where in f_j^k the lower index *j* represents the function

name index and the upper index k represents the record type: + for an entry, − for an exit, and number for a probe point k. 2-words will be represented accordingly by the set $\{f_1^+ f_2^+, f_2^+ f_3^+, f_3^+ f_3^1, f_3^1 f_3^2, f_3^2 f_3^-, f_2^- f_1^-, f_3^- f_2^-\}$. All 1- and 2- words are unique in this case – their frequencies are equal to 1.

Additional measures, such as Entropy measures [6] or N-stacks , can be introduced as needed. If a user is interested in calculating *exact* matching between traces, then hashing techniques can be used for compression. However, these techniques are not applicable for approximate trace matching, since no non-trivial measure of distance can be established between hashed traces.

2.3.2.1.2 Algorithm for Comparing a Pair of Processes

We summarize the algorithm for comparing a pair of processes at different levels of compression in Figure 2. Functional forms of $M_k(X,Y)$ are given below.

Suppose we compare fingerprints of two processes p_1 and p_2 at the level of compression l. The distance between two processes is denoted by d.

Procedure compare_processes (p_1, p_2, l).
  If (level l fingerprint contains frequency info)
    $d \leftarrow M_l(p_1, p_2)$;
  else if (level l fingerprint does not contain frequency info)
    $d \leftarrow M_l^*(p_1, p_2)$;
  else if (level l represents uncompressed trace)
    $d \leftarrow M^\bullet(p_1, p_2)$;
  return d;

Figure 2. Algorithm for comparing two processes

In order to measure the distance $M^\bullet(X,Y)$ between two uncompressed processes X and Y, processes are represented as strings; string comparison algorithms, such as diff [26] or Pattern Hunter II [19], can be used to find similarities and differences. As a measure of

distance between two uncompressed traces we use the Levenshtein distance[8] [18], denoted by M^U. An example of comparing two processes is given in Figure 3.

For comparing two compressed processes X and Y at the level of compression k (see Section 2.3.2.2 for details), we use the following metric:

$$M_k(X,Y) = \begin{cases} \sum_{i}^{X \cup Y} \lambda[X(i), Y(i)], \text{ if } X \cap Y \neq \varnothing \\ \infty, \text{ if } X \cap Y = \varnothing \end{cases}, \qquad (2.1)$$

where $\lambda(\alpha,\beta) = \dfrac{\max(\alpha,\beta)+1}{\min(\alpha,\beta)+1} - 1,$ and $X(i)$ and $Y(i)$ will return the value of the i-th member in the set, e.g., frequency of occurrence, or zero if the member is absent. The summation is performed for all elements of sets X and Y ($X \cup Y$). We add 1 to both the numerator and the denominator to avoid division by zero and subtract one from the ratio to get 0 if $X(i) = Y(i)$.

$$p_1 = f_1^+ f_3^+ f_3^1 f_3^- f_1^-,$$
$$p_2 = f_1^+ f_2^+ f_3^+ f_3^2 f_3^- f_2^- f_1^-,$$
$$p_1 \leftrightarrow p_2 = f_1^+ \overline{f_2^+} f_3^+ \overline{f_3^1} \underline{f_3^2} f_3^- \overline{f_2^-} f_1^-.$$

Figure 3. Comparison of two uncompressed processes. Upper dashes depict functionality present only in Process 1; lower dashes – functionality present only in Process 2; no dashes – common functionality.

Note that the usage of p-norms, e.g., Euclidean norm is not desirable; since they will not highlight information about non-overlapping set members, while metric (2.1) will; see Example 1 for details.

---

[8] Levenshtein distance between two strings is given by the minimum number of operations (insertion, deletion, and substitution) needed to transform one string into another.

Example 1.

Suppose we have two processes: *A* containing components *a* and *b* with frequencies 4 and 3 accordingly; *B* containing components *a*, *b*, and *c* with frequencies 5, 3 and 1; and *C* containing components *a*, *b*, and *c* with frequencies 4, 3 and 2. We can write these fingerprints as three vectors: A = [4 3 0], B = [5 3 1], and C=[4 3 2].

By calculating the distance between processes[9] ($\|A-B\|_2$ denotes Euclidean distance between *A* and *B*) we get

$$\begin{aligned} M(A,B) &= \sum [0.2 \quad 0 \quad 1] = 1.2, \\ M(B,C) &= \sum [0 \quad 0 \quad 1] = 1.0, \\ \|A-B\|_2 &= \sum [1 \quad 0 \quad 1] = 2.0, \\ \|B-C\|_2 &= \sum [1 \quad 0 \quad 1] = 2.0. \end{aligned} \tag{2.2}$$

Euclidean norm treats all dimensions equally; it shows that the distance between processes *A* and *B* is the same as between processes *B* and *C*. However, we want to emphasize the fact that *A* and *B* are further apart than *B* and *C*, since component *c* is missing in *A* and metric *M* highlights this fact.

For those fingerprints that do not contain frequency information (vectors whose *i*-th value is 1 when the element is present in the uncompressed trace and 0 otherwise), we can use a simple metric

$$M_k^*(X,Y) = \begin{cases} 0, X \cap Y \neq \varnothing \\ \infty, X \cap Y = \varnothing \end{cases}. \tag{2.3}$$

This measure is conservative, but is rather fast to compute

---

[9] Measure (2.1) is calculated as *M*(*A*,*B*) = [(max(4,5)+1)/(min(4,5) + 1)-1]+ [(max(3,3)+1)/(min(3,3) + 1)-1]+ [(max(0,1)+1)/(min(0,1) + 1)-1] = (6/5-1)+(4/4-1)+(2/1-1)=1.2

2.3.2.1.3 Algorithm for Comparing a Pair of Traces

Since the traces consist of multiple processes, we need to perform cross-comparisons between each pair of processes and aggregate this data to obtain a quantitative description of the distance between traces. To do this, we need a (possibly heuristic) distance measure that calculates the distance between a pair of similar traces to be less than the distance between a pair of less similar traces.

No simple one-dimensional measure of a complicated concept such as the "difference between two traces" can capture all desired features. Is *Moby Dick* closer to the Book of Genesis than it is to the *Catcher in the Rye*? It is unlikely that a simple heuristic can be devised that answers this question to everyone's satisfaction. Nevertheless, we need a heuristic. Our heuristic is defined in Figure 4.

2.3.2.1.4 The Overall Approach

The algorithms implement SIFT through trace and process comparison, as well as fingerprinting. During software execution, in a concurrent processing environment, a trace t could consist of the multiple, parallel, processes executed during the software run. If the total number of processes in a given trace t is equal to m then

$$t = \{p_1, p_2, \ldots, p_m\}.$$

Later, we will use this information to compute the similarity between given traces. In general, a process may split into multiple *threads*. For the sake of simplicity, in this paper we assume that each process consists of a single thread – henceforth processes and threads are treated as equal. Comparing two multi-threaded processes is analogous to comparing two traces with multiple processes.

Suppose we compare two traces t_1 and t_2 with number of processes equal to N and M respectively at the level of compression l. Distances between processes of t_1 and t_2 are calculated and stored in matrix D with m rows and n columns (defined by the conditions below). Function $t(k)$ returns k-th process of trace t. The distance between

two traces is denoted by d.

```
Procedure compare_traces (t1, t2, l)
        //fill in the distance matrix
        for i ← 1 to N
                for j ← 1 to M
                  D(i,j) ← compare_processes (t1(i), t2(j), l);
        //calculate the distance between traces
        (d1, p1) ← calc_trc_d(D);
        (d2, p2) ← calc_trc_d(transpose D);
        d ← (d1 + d2)/2,
  //percentage of non-overlapping processes
        p ← (p1 + p2) / (N + M);
        return (d, p);
//calculate distance between pair of traces
Procedure calc_trc_d(D)
        d ← 0; //distance between overlapping processes
        p ← 0; //number of non-overlapping processes
        for each row in D
                m the minimal distance in a given row;
                if d = ∞
                        p++;
                else
                        d += m;
        return (d, p);
```

Figure 4. Algorithm for measuring distance between traces.[10]

---

[10] Note that we can speed up comparison process by keeping track of similar pairs of processes from higher levels of compression.

Let the set of all possible elements of contextual information that can be captured during software execution be denoted by E. A process p with n elements is represented by a sequence

$$p = e_i\Big|_{i=1}^{N} \qquad (2.4)$$

where i indexes the ith event captured, and each event $e \in E$. In order to compute the distance between two given traces, we need to compute that measure in terms of the distance between the respective sets of processes contained within the two traces. In essence, we perform cross-comparison between each pair of processes (one process from each of the two given traces) to obtain their distance, and aggregate this data to obtain a quantitative measure of the distance between the two given traces. The distance between two given processes is computed as follows:

compare_processes(prc_id1, prc_id2, c_l) → d_p

where, prc_id1 and prc_id2 are processes at the level of compression c_l; and the comparison process returns a measure of distance between processes d_p. Compare_processes is outlined in Section 2.3.2.1.2.

The process information can be compressed (using lossy compression techniques, i.e., compression with loss of information) into the "fingerprints" introduced in Section 2.3.2.1.1. Individual uncompressed processes are independently compressed, using different compression formulae, to obtain various compression levels. The level of compression of a process specifies the type of fingerprint that should be used in the comparison procedure.

Based on process comparison, we can now describe how the traces are compared. The comparison procedure within any given context considered in Section 2.3.2.1 is defined as follows:

compare_traces(trc_id1, trc_id2, c_l) → {d_t, p_p},

where trc_id1 and trc_id2 are execution traces at the level of compression c_l; and the output of this comparison is given by a tuple of distance measure between traces d_t and percentage of non-overlapping processes p_p. The parameter c_l is set by the analyst and is, in fact, passed on to the compare_processes procedure discussed above. Details of compare_traces are given in Section 2.3.2.1.3.

2.3.2.2 The Iterative-Unfolding Algorithms

As described in Section 2.3.1, the traces are compared iteratively from the highest to the lowest level of compression. During each iteration, the comparison follows the procedure described in Section 2.3.2.1.4. Recall from Section 2.3.2 that there are two practical contexts for comparing traces: (i) a single trace against a set of traces and (ii) within a given set of traces. Thus, Figure 5 and Figure 6 respectively describe the two algorithms to deal with these contexts.

Assume that compression level l is in the range [0, 1, …, N], where 0 is an uncompressed trace level and N is a fingerprint containing the least amount of information. A single trace t is compared against a set of traces S. The array T_d (of size N+1) contains maximum measures of distance between traces for different compression levels. Maximum percentage [0,1] of non-matching processes is denoted by T_p. The end result is a set of traces below the threshold level.

Procedure $P_1(t, S, T_d, T_p)$

```
for l = N to 0
    for each trace in S
        (d, p) ← compare_traces(t, trace, l) ;
        if d > Td(l) or p > Tp
            S = S – trace;
return S;
```

Figure 5. Algorithm for comparing a single trace *t* against a set of traces *S*.

The traces are clustered using the Agglomerative Hierarchical Clustering (AHC) algorithm [14]. It is computed by a function cluster(d_m,d_t),which takes the distance matrix between traces d_m and the maximum distance between traces to be clustered d_t. The distance between clusters is determined by measuring the maximum distance between elements of each cluster. The clustering is stopped when, based on a particular distance criterion[11], the clusters are too far apart to be merged.

2.4 Analysis

In this section we analyze the algorithms presented in Section 2.3.2. The efficiency analysis of the functions is described in Section 2.4.1. Section 2.4.2 describes constraints within which the algorithms are accurate. Section 2.4.3 describes the overhead due to our approach.

2.4.1 Efficiency

In Section 2.4.1.1 we discuss efficiency of the core algorithms. Efficiency of the Iterative-Unfolding Algorithms is shown in Section 2.4.1.2. Special cases are discussed in Section 2.4.1.3.

2.4.1.1 Core Algorithms Efficiency

The derivation of our algorithms asymptotic behaviour may be found in [24]. Only the final results are presented here. Look at the maximum number of algorithm operations required to compare a single trace *t* against a set of traces *S* (given in Figure 5). In the worst-case scenario, when all traces within *S* are close to *t* and cannot be filtered out, we will have to perform all iterations on the full set |S|:

$$C^{\max}[P_1(t,S)] \in \underbrace{\mathcal{O}\left(K\,|\,S\,|\,\widehat{N}^2\widehat{L}_l\right)}_{\alpha} + \underbrace{\mathcal{O}\left(|\,S\,|\,\widehat{N}^2\widehat{L}_0^2\right)}_{\beta} \overset{K\widehat{L}_l<\widehat{L}_0^2}{=} \mathcal{O}\left(|\,S\,|\,\widehat{N}^2\widehat{L}_0^2\right) \qquad (2.5)$$

---

[11] Given two clusters *A* and *B*, the distance is calculated by max{compare_traces $(a, b, l) : a \in A, b \in B$}.

where K is the maximum level of compression, $\hat{N}$ is the maximum possible number of processes in a trace and $\hat{L}_l$ is the maximum length of a given fingerprint at compression level l. The comparison for $l > 0$ is performed using measures of distance (2.1) and (2.3); comparison of uncompressed traces is performed at $l = 0$ (using diff[26]), see Section 2.3.2.1.2 for details. The term α arises from comparison of fingerprints and the term β from comparison of uncompressed traces. Note that, in practice, most traces will be filtered out at high levels of compression and that the length of fingerprints, by construction, should be small.

The algorithm for comparison within a set of traces calls a recursive procedure "compare" (given in Figure 6). The running time, T, of the function "compare" can be represented as

$$T(|S|) = \sum_{i=1}^{a} T\left(\lfloor |S|/b_i \rfloor\right) + \text{compare}(\mathcal{P}_S), \qquad (2.6)$$

where $\mathcal{P}_S$ is a set of properties of traces in set S. Coefficients b_i (fraction of elements in S) and a (number of clusters) change at each iteration − they are obtained from the clustering procedure (called from "compare") and will depend on $\mathcal{P}_S$.

The problem formulated in (2.6) is too general and, to the best of our knowledge, cannot be "unraveled" without knowing distributions of the parameters b_i and a. The worst-case scenario is when a=1 and b_i=1, i.e., members of the set S cannot be partitioned into subsets since they are too close to each other. In this case

$$C^{\max}[P_2(S)] \in \mathcal{O}\left(|S|^2 \, K\hat{N}^2\hat{L}_l\right) + \mathcal{O}\left(|S|^2 \, \hat{N}^2\hat{L}_0^2\right) \overset{K\hat{L}_l < \hat{L}_0^2}{=} \mathcal{O}\left(|S|^2 \, \hat{N}^2\hat{L}_0^2\right). \qquad (2.7)$$

In closing, the worst-case scenario computational time for $P_1(t,S)$ grows linearly with the number of traces, and quadratically with the number of processes per trace and the length

of traces. The algorithm $P_2(T,S)$ is quadratic in the number of traces, the number of processes per trace and the length of traces.

Assume that compression level l is in the range [0, 1, … , N], where 0 is an uncompressed trace level and N is a fingerprint containing the least amount of information. The initial set of traces is given by trace set S. $S(i)$ returns the i-th member of the set S. Array T_d (of size N+1) contains the maximum measures of distance between traces for different compression levels. Maximum percentage [0,1] of non-matching processes is denoted by T_p. Global variable S_{out} stores similar clusters of traces.

```
//Transform 2-tuples distance measure into a scalar
Procedure condense_tuple(d, p, T_p)
        if p > T_p
                m ← ∞;
        else
                m ← d;
        return m;
//recursive comparison function (note that recursion can be //"unraveled" by parsing
the tree in breadth)
Procedure compare (trace_set, l, T_d, T_p)
        //calculate distance matrix D between traces
        M ← cardinality(trace_set);
        for i=1 to M
                for j=i+1 to M //D is symmetric
                        (d, p) ← compare_traces(trace_set(i), trace_set(j), l);
                        D(i, j) ← condense_tuple(d, p, T_p);
        //Cluster traces using distance data in D
        clusters ← cluster(D, T_d(l));
        if (l > 0)
                for each cluster in clusters
```

```
                    compare(cluster, l – 1, T_d, T_p)
        else //reached uncompressed level
                add trace_set to S_out;
//main procedure
Procedure P_2(S)
        Set T_d and T_p;
        compare (S, N, T_d, T_p);
        return S_out ;
```

Figure 6. Algorithm for comparing traces within a given set *S*.

2.4.1.2 Special Cases

We identify two cases where the direct comparison approach is more efficient than the iterative-unfolding approach. The first case occurs when the traces of interest are very similar; the traces won't be filtered out at higher levels of comparison and so direct comparison becomes necessary at the uncompressed level. The second case occurs when the traces are small (i.e., the length of the processes in the traces is comparable to the length of the fingerprints) and the traces consist of only a few processes so comparison times between the iterative-unfolding approach and uncompressed comparisons would be more or less equivalent. Note that if the number of processes is large, our approach may yield superior results by aggregating information from different processes into a single fingerprint. In all other cases, our approach is superior.

An additional case occurs when one needs to identify identical traces (e.g., for identification of duplicate test cases). Two iterations are needed in this case. The first one uses hashes of processes as fingerprints. The second iteration is needed to verify the result of first iteration by analyzing uncompressed traces (the Levenshtein distance [18] between processes should be equal to 0).

2.4.2 Method Accuracy

In order for the iterative-unfolding approach to be accurate, similar traces must not be accidentally discarded at high levels of compression (we throw away all traces farther apart than a level-specific threshold). To accomplish this, we must carefully select the threshold's values for distance measures[12] between traces. General guidance is given by the following conjecture (axiomatic in nature).

> We conjecture that for every distance threshold $T_{k+1} > 0$ at level $k + 1$ there exists another threshold $T_k > 0$ for level k, such that if $d_{k+1}(A,B) > T_{k+1}$ then $d_k(A,B) > T_k$. (where $d_j(A,B)$ represents the measure of distance between traces A and B at compression level j). This conjecture is reasonable but we have not proved it, nor do we have an explicit way to compute T_k from T_{k+1} except in some special cases as detailed below. Traces A and B are considered dissimilar at compression level j if $d_j(A,B) > T_j$. If they are not dissimilar, they are considered to be similar. Our conjecture now states that if two strings are dissimilar at a high level of compression, they will also be dissimilar at the corresponding lower level of compression. For the example of strings with N letters, each taking two values, where lexical distance d_0 is between 0 and N and letter count distance d_1 is also between 0 and N, $T := T_1 = T_0$. In this case, it follows that if two compressed strings are dissimilar at level 1 with threshold T, they must also be dissimilar at uncompressed level 0 at the same threshold T.

If this conjecture holds, an analyst can specify the desired zero compression threshold T_0 and use the conjecture backwards to generate thresholds for all other levels $T_1, \ldots, T_K$, where K is the highest level of compression[13].

---

[12] Examples of measures of distance are given in Section 2.3.2.1.2.

[13] In the trivial case of finding identical traces the measure of distance will be equal to 0 for all levels of compression. Note there is no general way to determine a bound on the fingerprint distance given a bound

The iterative-unfolding algorithms (described in Section 2.3.2) can then be applied to traces of interest (starting at level *K*). There will be no false removals of similar traces assuming the conjecture holds.

Dissimilar traces are rapidly rejected. Clearly there will be special cases when our method will fail to filter out distant traces. For example, consider a pair of strings "aabb" and "bbaa". The character frequencies of these two strings are identical, so any *p*-norm distance or our metrics (2.1) and (2.2) will show that these two strings are identical at the 1-word level (sequences of strings of length 1). However, the Levenshtein distance between the uncompressed strings will take the largest possible value of 4. Note that the fraction of such string pairs from the total number of permutations decreases rapidly with the growth of string length and dictionary size (number of distinct words), i.e., such events should be rare. For example, percentage of non-filtered strings of length N=2 and letter dictionary of size d=2 is 12.5%; string of N=2 and d=4 is 4.69%; string of N=6 and d=2 is 0.05%; and strings of N=6 and d=4 is 0.04%.

2.4.3 Iteration-unfolding overheads

Prior to comparing traces, there is a need to generate the fingerprints, which consumes time and storage space. However, the benefit of the proposed approach would outweigh the time overhead because in a test environment we would expect that fingerprinted traces would be repeatedly used for comparison purposes. While the overhead due to storage does exist, we anticipate that this would not be too significant.

---

on the final edit-distance metric, since strings of any and all final edit-distance metrics are mapped into compressed fingerprints with any and all compressed level distances.

2.5 Implementation

With reference to Figure 1, such traces can be automatically produced by suitably instrumenting the application software. During software execution, multiple processes invoking different threads can run in parallel. Thus, a trace can consist of multiple execution paths collected from the individual processes executed during the software run.

The current implementation uses ten levels of compression. The lowest level of compression is denoted by L0; the highest by L9. For a given trace, we have:

L0. Uncompressed trace,

where each process within the trace is represented by a separate fingerprint. We now condense the fingerprinted information for each process in L0 to obtain, independently, levels L1 to L5:

L1. List of 2-words[14] with frequency information (FI),

L2. List of 1-words with FI,

L3. List of function names with FI,

L4. List of components with FI,

L5. List of components without FI.

Finally, the fingerprinted information from different processes at level L0 is merged into a single fingerprint per trace. This speeds up comparison, because pair-wise comparison of processes reduces to a single comparison for each pair of traces.

L6. Merged list of function names with FI,

---

[14] An *l*-word represents a continuous substring of length *l* from a string. A trace can be represented as a string, where each trace element is a character; see Section 3.2.1.1 for details.

L7. Merged list of function names without FI,

L8. Merged list of components with FI,

L9. Merged list of components without FI.

We use M^U measure of distance for L0, $M_k(X,Y)$ for levels with frequency information, and $M_k^*(X,Y)$ for levels without frequency information. The choice of the number of compression levels depends on such factors as: the number of levels of abstractions of traces considered appropriate due to variety and complexity of program constructs, application domain complexity, and the size of the trace dataset. In essence, the greater the variability in traces, the greater the chance that a higher number of compression levels are required. In our study, we decided to have ten levels. The comparison process, including clustering, is implemented based on the algorithms described in Section 2.3.2.2. The tool's prototype is implemented in Perl.

2.6 Validation Case Study

Our experimental ground is the complex system alluded to Section 2.1. Out of hundreds of thousands of test cases (from all phases of testing) that in some cases have a legacy of more than ten years, we gathered 1416 multithreaded tests cases from 21 test suites covering various features of the SUS[15]. An internal capturing facility gathers and stores execution paths and then outputs them in the format shown in Figure 1.

The captured 1416 traces consist of 73763 processes. Descriptive statistics of the distributions of process-length, trace-length and the number of processes per trace are given in Table 3.

[15] While there are thousands of execution traces in the SUS trace-database, it takes time to prepare them for trace analysis (e.g., due to analysing and modifying individually the test-suite scripts). This is an on-going process in our research to gather more traces.

We created fingerprints for the traces using the 10 levels of compression described in Section 2.5. The total space overhead is 9.1% (of the space needed to store the raw traces), which makes it feasible to store the fingerprints.

Table 3. Descriptive statistics of traces

Distribution of	Min	Mean	Max
number of processes per trace	2.00E0	5.21E1	1.41E3
process-length	1.00E0	3.70E4	1.51E8
trace-length	4.26E2	1.93E6	1.55E8

To validate the comparison of *a single trace against a set of traces*, we randomly picked a trace from our experimental set of 1416 traces and compared it against a random subset of traces from the experimental set. Such comparisons were performed 450 times (i.e., 30 draws of subsets of different sizes: 10 to 1410 traces with a step-size of 100). All computations were performed on an Intel Xeon™ 5160, 3.0 GHz Dual Core computer.

The execution time results are shown in Figure 7, where the different trace subset-sizes (10, 110, 210, etc.) are shown on the X-axis and the execution times on the Y-axis. We see that the computational time for each draw (denoted by circles) splits into two patterns: the first, where computation time is less than 5 seconds, and the second, where the computational time is of the order of 250 seconds. The first pattern (lower part of the figure) represents the situation when there are no similar traces in the reference subset; and, therefore, all traces are eliminated at high levels of compression. The second pattern (upper part of the figure) represents the situation when the residual traces in the reference subset (after elimination) are the same (or very similar to one another). Since these traces are at lower levels of compression, the comparison times are high. The frequency-counts of the number of comparisons (totalling 30 for any subset-size) are shown with the lines (right vertical axis). The solid line shows the number of comparisons in the lower pattern, while the dotted line shows this in the upper pattern. Note that for the first pattern, the execution time grows linearly with the increase of the subset-sizes ($R^2 = 0.998$ and *p*-value $< 2.2 \times 10^{-16}$). However, for the second pattern, even though execution times are constant, the variance of 5.4 seconds in execution times masks the linear growth of

approximately 0.4 seconds over 100 traces. Because the algorithm performs pair-wise comparison of processes at the higher levels of compression (L0-L5), we might expect execution time to grow cubicly (the number of operations is proportional to the number of traces multiplied by the number of pair-wise process comparisons). However, since most of the traces are filtered out at the higher levels of compression, where information about traces is discarded (L6-L9), the complexity is reduced to linear.

Figure 8 and Figure 9 show the results of comparing *within the given set of traces* (i.e., clustering). The experimental set-up here is similar to the one described earlier. Figure 8 shows, based on regression analysis, that the execution time for clustering grows quadratically ($R^2 = 0.991$ and p-value $< 10^{-16}$) with the growth of the subset-sizes (≈23.0 seconds per 100 traces). As with the first experiment, the influence of pair-wise thread comparison is negligible, since most of the traces are discarded at the higher levels of compression (L6-L9). Therefore, comparison time grows quadratically and not quartically (regression analysis confirms this observation). Figure 9 shows that 85 to 90 percent of traces are filtered out after six iterations (start to L4) of clustering. Again, this demonstrates that the approach is scalable. When we compare these results with the results of the case study that we performed on a dataset of 116 traces [24], we see that additional levels of compression are needed with increase of the dataset size. In [24] most of the traces were filtered out after 4 iterations (start to L6), majority of the clusters were eliminated by that time as well.

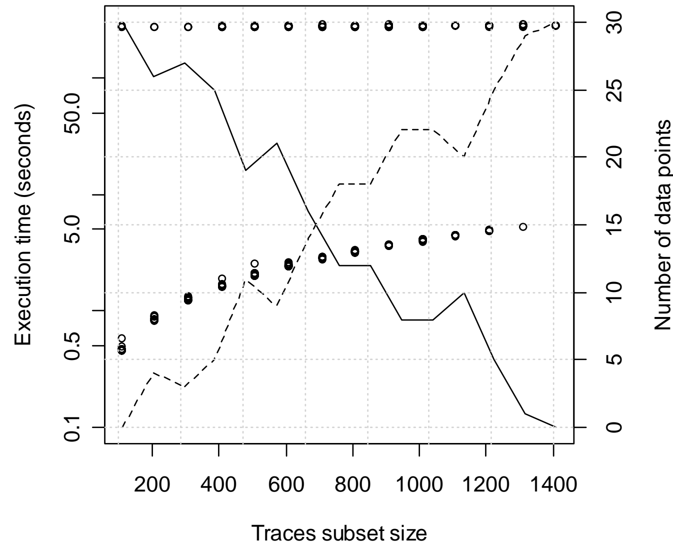

Figure 7. Timing of comparing a trace against a set of traces; timing for each draw is denoted by circles; values are plotted on the left axis. Solid line shows the number of comparisons in the lower pattern; dotted line in the upper pattern; values of both lines are plotted on the right vertical axis.

If we assume that the behavior of our algorithms will hold for larger datasets, then we can predict the time needed to perform computations for large values of |S| by extrapolating fitted regression.

The results of extrapolation for the algorithm $P_1(t,S)$ are given in Figure 10a. In order to compare a single trace with a set of 10,000 traces, an analyst will need ≈39 seconds (on average) to filter out dissimilar traces, plus an additional 250 seconds for comparing every trace in S so similar to t that it should be compared with t at the uncompressed level. The typical and frequent situation for application of this algorithm is when a defect is found in the field, and the analyst, upon obtaining a UT from the user, needs to compare it against all HTs to identify missing coverage where a defect may potentially reside.

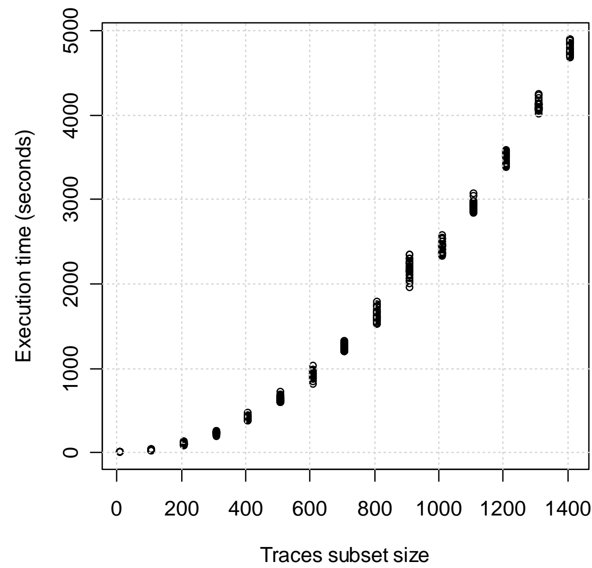

Figure 8. Timing of comparing traces within a given set

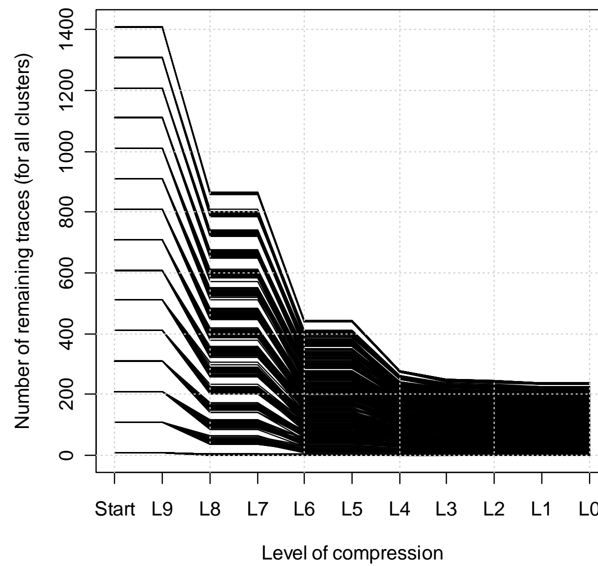

Figure 9. Number of traces remaining after each iteration of comparing traces within a given set

Figure 10b shows the extrapolated data for the algorithm $P_2(S)$. In order to perform filtering within a set of 10,000 traces, an analyst will need 2.3×10^5 seconds ($\approx$2.6 days)[16]. This type of analysis is useful for customer profiling, identification of redundant test cases, etc. Note that such analysis is typically infrequent in a production environment.

As described in the Introduction section, our attempt to experiment with the kBehaviour prototype [15] clearly indicated that the performance of the SIFT algorithm is better than that of the kBehaviour approach.

We also compared the execution time of comparing a pair of traces using our technique and comparing the same pair of traces using diff [26]. Processes within traces were compared pair-wise. Traces were represented using text files with each trace record stored on a separate line. The SIFT approach significantly outperforms diff. For example, the comparison of a medium size trace T1 (113 processes, $\approx 10^6$ elements) with a small trace T2 (40 processes, $\approx 10^3$ elements) took 0.02 seconds with our iterative-unfolding technique and 1476 seconds with diff. The comparison of T1 with another medium size trace T3 (147 processes, $\approx 10^6$ elements) took 0.05 seconds using our technique and 12635 seconds with diff. Further tests were just as confirmatory. Thus, we can see that direct comparison using diff does not scale. This can be explained two ways. First, the complexity of comparing two different strings with diff is quadratic [26]. Second, since we need to compare each pair of processes independently, the complexity increases quadratically with an increase in the number of processes.

---

[16] Note that our prototype is implemented in Perl; a production version in C/C++ would be faster. Additional increase in speed should result from parallelisation of the algorithms.

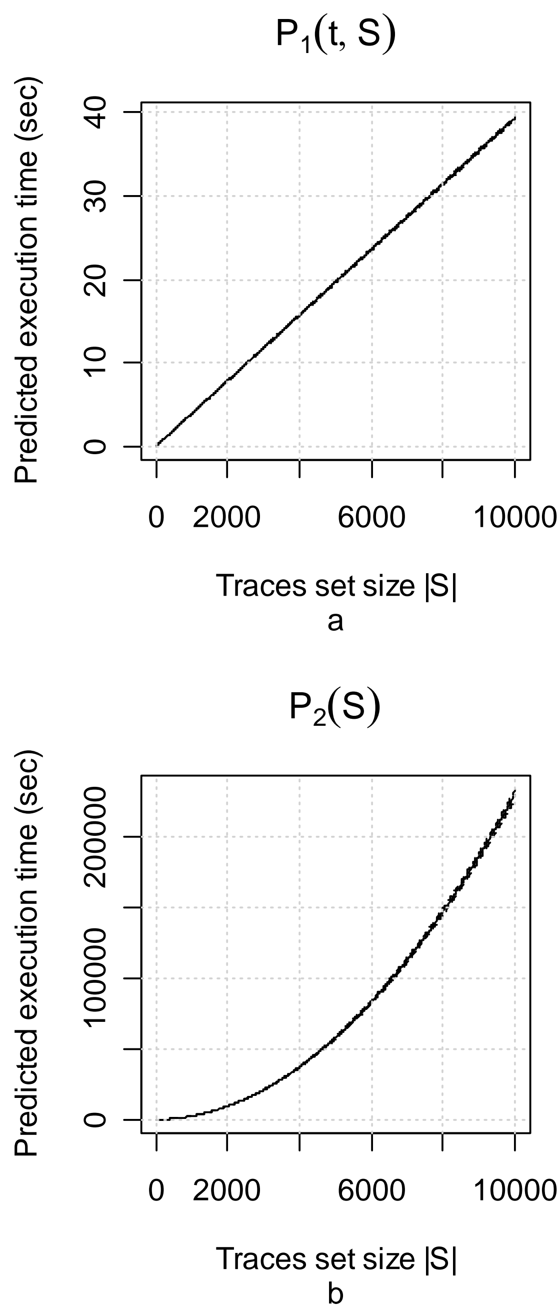

Figure 10. Predicted time (dotted lines represent 95% confidence bands) for (a) $P_1(t,S)$ – linear, and (b) $P_2(S)$ – quadratic, based on extrapolation of the fitted regression.

2.7 Conclusion And Future Work

The comparison of traces resulting from the execution of a software system is of considerable interest for a variety of purposes, such as software testing [1-3], program comprehension [13-15], and security [11]. In this paper, we have proposed a new, iterative-unfolding, approach (called SIFT) for filtering-out traces to help speed up the overall comparison process.

The essence of this approach is that it iteratively compares traces at different levels of compression, from high to low, and in the process it rapidly eliminates dissimilar traces, eventually leaving residual, similar, traces at the lowest level of compaction. Once similar traces are identified, they can be passed to external tools for further analysis. We use fingerprinting techniques for compressing traces, and comparison and clustering algorithms for identifying similar traces.

Our approach can be packaged as a framework, where the component algorithms and techniques can be replaced with alternate techniques making the framework portable to other development environments. Further details are web-accessible from [24, Section 5], where also details of the usage environment of this technology are described (see [24, Section 6]).

The paper describes a significant case study involving 1416 traces from a large, distributed software system in use at numerous sites worldwide. The efficiency of the approach is linear with the growth of the number of traces when comparing a trace against a set of traces, and quadratic when comparing within a set of traces using clustering techniques (see Figure 10). The timings from the case study data are feasible in a practical environment. From these results, we thus conclude that the iterative-unfolding approach is scalable for use in a practical environment

We plan to conduct a number of further case studies. These include, for example, increasing the dataset size; validation of set-to-set comparisons; and cost-benefit analysis

in a practical environment. Our tool development effort is on-going with the long-term goal being to transfer the technology to the production environment.

References

[1] Avvari, M. V., Chin, P. A., Nandigama, M. K. and Dhanikonda, *Software application test coverage analyzer*. U.S. Patent #6,978,401, Sun Microsystems, Inc., U.S., (2005).

[2] Biermann, A. and Feldman, J. On the Synthesis of Finite State Machines from Samples of their Behavior. *IEEE Trans. Computers*, 21, 6 (1972), 592–597.

[3] Cook, J. E. and Wolf, A. Discovering Models of Software Processes from Event-Based Data. *ACM Trans. Software Eng. and Methodology*, 7, 3 (1998), 215-249.

[4] Cotroneo, D., Pietrantuono, R., Mariani, L. and Pastore, F. Investigation of Failure Causes in Workload-Driven Reliability Testing. In *Proc. 4th International Workshop on Software Quality Assurance* (2007), 78-85.

[5] Dallmeier, V., Lindig, C. and Zeller, A. Lightweight Defect Localization for Java. In *Proc. European Conference on Object Oriented Programming* (2005), 528-550.

[6] Davison, M., Gittens, M., Godwin, D., Madhavji, N. H., Miranskyy, A. V. and Wilding, M. *Improvement of computer software test coverage analysis.* . U.S. Patent Application # 11/549410, IBM Corp., (2006).

[7] Elbaum, S., Kanduri, S. and Andrews, A. Trace anomalies as precursors of field failures: an empirical study *Empir. Software Eng.* , 12, 5 (2007), 447-469.

[8] Elbaum, S., Rothermel, G., Kanduri, S. and Malishevsky, A. G. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Control*, 12, 3 (2004), 185-210.

[9] Feder, T. and Motwani, R. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Syst. Sci.*, 51, 2 (1995), 261-272.

[10] Fremuth-Paeger, C. and Jungnickel, D. Balanced network flows. VIII. A revised theory of phase-ordered algorithms and the O(sqrt(n) m log(n^2 /m)/log n) bound for the nonbipartite cardinality matching problem. *Networks*, 41, 3 (2003), 137-142.

[11] Greevy, O., Ducasse, S. and Girba, T. Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis. In *Proc. 21st IEEE Int'l Conference on Software Maintenance* (2005), 347-356.

[12] Hamou-Lhadj, A. and Lethbridge, T. C. Compression techniques to simplify the analysis of large execution traces. In *Proc. 10th Int'l Wkshp on Program Comprehension* (2002), 159-168.

[13] Haran, M., Karr, A., Orso, A., Porter, A. and Sanil, A. Applying classification techniques to remotely-collected program execution data. *SIGSOFT Softw. Eng. Notes*, 30, 5 (2005), 146-155.

[14] Jain, A. K., Murty, M. N. and Flynn, P. J. Data clustering: a review. *ACM Computing Surveys*, 31, 3 (1999), 264-323.

[15] kBehavior http://www.lta.disco.unimib.it/kbehavior/.

[16] Kuhn, A. and Greevy, O. Exploiting the Analogy Between Traces and Signal Processing In *Proc. 22nd IEEE Int'l Conference on Softw. Maintenance* (2006), 320-329.

[17] Lee, W., Stolfo, S. J. and Chan, P. K. Learning patterns from unix process execution traces for intrusion detection. In *Proc. AAAI Workshop: AI Approaches to Fraud Detection and Risk Management* (1997), 50-56.

[18] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions, and reversals (Russian). *Doklady Akademii Nauk SSSR*, 163, 4 (1966), 845-848.

[19] Li, M., Ma, B., Kisman, D. and Tromp, J. PatternHunter II: Highly Sensitive and Fast Homology Search. *J. of Bioinformatics and Computational Biology*, 2, 3 (2004), 417-440.

[20] Mariani, L. and Pezzè, M. *Inference of component protocols by the kBehavior algorithm*. University of Milano Bicocca, 2004.

[21] Mariani, L. and Pezzè, M. Dynamic Detection of COTS Component Incompatibility. *IEEE Software*, 24, 5 (2007), 76-85.

[22] Masri, W., Podgurski, A. and Leon, D. An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. *IEEE Trans. Softw. Eng.*, 33, 7 (2007), 454-477.

[23] Miranskyy, A. V., Gittens, M. S., Madhavji, N. H. and Taylor, C. A. Usage of Long Execution Sequences for Test Case Prioritization. In *Proc. Suppl. Proc. of 18th IEEE Int'l Symp. on Softw. Reliability Eng.* (2007).

[24] Miranskyy, A. V., Madhavji, N. H., Gittens, M. S., Davison, M., Wilding, M. and Godwin, D. *An Iterative, Multi-Level, and Scalable Approach to Comparing Execution Traces, TR-74.209*. IBM Center for Advanced Studies (CAS), Toronto, 2007 (https://www.ibm.com/ibm/cas/publications/index.shtml).

[25] Moe, J. and Carr, D. A. Using execution trace data to improve distributed systems. *Softw. Pract. Exper.*, 32(2002), 889-906.

[26] Myers, E. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1, 2 (1986), 251-266.

[27] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. and Wang, B. Automated support for classifying software failure reports. In *Proc. 25th Int'l Conference on Software Engineering* (2003), 465 - 475.

[28] Reiss, S. P. and Renieris, M. Encoding program executions. In *Proc. 23rd Int'l Conf. on Software Engineering* (2001), 221-230.

[29] Renieris, M., Ramaprasad, S. and Reiss, S. P. Arithmetic program paths. In *Proc. 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), 90-98.

[30] Rothermel, G., Harrold, M. J., Ostrin, J. and Hong, C. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *Proc. International Conference on Software Maintenance* (1998), 34-43.

[31] Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M. and Ma, W.-Y. Automated known problem diagnosis with event traces. In *Proc. 2006 EuroSys conference* (2006), 357-388.

Chapter 3

3 Using Entropy Measures for Comparison of Software Traces

The analysis of execution paths (also known as software traces) collected from a given software product can help in a number of areas including software testing, software maintenance and program comprehension.

In this chapter, we study the applicability of Shannon entropy and three extended entropies (Landsberg-Vedral, Rényi, and Tsallis) to the classification of traces related to various software defects. Our validation study shows the three extended entropies, with parameters chosen to emphasize rare events, show good performance.

3.1 Introduction

A software execution trace can be thought of as a log of information captured during any particular execution-run of software. For example, a trace in Figure 11 shows the program flow entering function f1; calling f2 from f1; f2 recursively calling itself, and, eventually, exiting these functions. In order to capture this information, each function in the software is instrumented to log entry and exit points to a function.

```
1            f1 entry
2            | f2 entry
3            | | f2 entry
4            | | f2 exit
5            | f2 exit
6            f1 exit
```

Figure 11. An example of a trace

The comparison of program execution traces is important for a number of problem areas in software development and use. In the area of testing, for example, such comparisons are used to: 1) determine how well user execution paths (traces collected in the field) are

covered in testing [3, 7, 29]; 2) detect anomalous behavior arising during a component's upgrade or reuse [14]; 3) map and classify defects [9, 18, 24]; 4) determine redundant test cases executed by one or more test teams [20]; and 5) prioritize test cases (to maximize execution path coverage with a minimum number of test cases) [8, 15]. Trace comparisons are used also in operational profiling (for instance, in mapping the frequency of execution paths used by different user classes) [29] and intrusion analysis (e.g., detecting deviations of field execution paths from expectations) [13].

For some problems, such as test case prioritization, traces gathered in a condensed form (such as a vector of executed function names or caller-callee pairs) are adequate [8]. However, for others, such as the detection of missing coverage and anomalous behavior using state machines, detailed execution paths are necessary [3, 14]. The time required for analyzing traces can sometimes be extremely important. For instance 1) a customer support analyst using traces to map a reported defect onto an existing set of defects to identify the problem's root cause and advise a customer on how to fix her problem, and 2) a development analyst working with the testing team to identify missing coverage that resulted in a field defect.

Many trace comparison techniques are not scalable [16, 5]. Based on our experience, support personnel of a large-scale industrial application with hundreds of thousands of installations can collect tens of thousands of traces per year. Moreover, a trace collected on a production system is populated at a rate of millions of records per minute.

The described need to compare traces, together with a lack of reliable and scalable tools for doing this, motivated us to investigate alternate solutions. To speed up trace comparisons, we propose that traces first be filtered out from the given set, rejecting those that are not going to match with the test cases, allowing just the remaining few to be compared for target purposes. The underlying assumption (based on our practical experience) is that most traces are not even close to being similar, just a few are similar, and only a very few are identical.

This strategy is implemented and validated in the **S**calable **I**terative-un**F**olding **T**echnique (SIFT) [16]. The collected traces are first compressed into several levels prior to comparing them. Each level of compression uses a unique signature, which we call a "fingerprint"[17]. Starting with the highest compression level, the traces are compared, and unmatched ones are rejected. Iterating through the lower levels until the comparison process is complete leaves only traces that match at the lowest (or uncompressed) level. The SIFT objective ends here. The matched traces can then be passed on to external tools for further analysis such as defect or security breach identification.

The process of creating a fingerprint can be interpreted as a map from the very high dimensional space of traces to a low, ideally one-dimensional, space. Simple examples of such fingerprints are 1) the total number of unique function names in a trace and 2) the number of elements in a trace. However, while these fingerprints may be useful for our purposes, neither are sufficient. The "number of unique function names" fingerprint doesn't discriminate enough: many quite dissimilar traces can share the same function names called. At the other extreme, the number of elements in the trace discriminates too much − traces which are essentially similar may have varying numbers of elements. The mapping should be such that projections of traces of different types should be positioned far apart in the resulting small space.

Using the frequency of the function names called is the next step in selecting useful traces. A natural one dimensional representation of this data is the Shannon information [21], mathematically identical to the entropy of statistical mechanics. Other forms of entropy information, obeying slightly less restrictive axioms, have been defined [1]. These extended entropies (as reviewed in [4] are indexed by a parameter q which, when $q = 1$ reduces them to the traditional Shannon entropy and which can be set to make them more ($q < 1$) or less ($q > 1$) sensitive to the frequency of infrequently called functions,

[17] The fingerprint of the next iteration always contains more information than the fingerprint of the previous iteration, hence the term *unfolding*.

improving the classification power of algorithms. Indeed, an extended Rényi entropy [19] with $q = 0$ returns the "number of unique function names" fingerprint, the Hartley entropy of information theory.

The entropy concept can also be extended in another way. Traces differ not only in which functions they call but in the pattern linking the call of one function with the call of another. As such it makes sense to collect not only the frequency of function calls, but also the frequency of calling given pairs, triplets, and in general l-tuples of calls. The frequency information assembled for these "l-words" can be converted into "word entropies", for further discriminatory power. In addition each record in a trace can be encoded in different ways (denoted as c) by incorporating various information such as a record's function name or type.

In this paper we study the applicability of the Shannon entropy [21] and the Landsberg-Vedral [12], Rényi [19], and Tsallis [22] entropies to comparison and classification of traces related to various software defects. We also study the effect of q, l, and c values on the classification power of the entropies. Note that the idea of using word entropies for classification problems in general is not a new contribution of this paper.

Similar work has been done to apply word entropies classification problems in bioinformatics [23] and in the analysis of natural languages [6]. However, to the best of our knowledge, no one has applied word entropies to compare software traces (although [28] suggested using Shannon entropy to measure the complexity of software traces).

The structure of the chapter is as follows: in Section 3.2 we define entropies and explain the process of trace entropy calculation. The way in which entropies are used for trace classification is shown in Section 3.3. A case study describing and validating the applicability of entropies for trace classification is shown in Section 3.4. Finally, Section 3.5 summarizes the chapter.

3.2 Entropies and Traces: definitions

In this section, we describe techniques for extracting the probability of various events from traces (Section 3.2.1) and usage of this information to calculate entropies of traces (Section 3.2.2).

3.2.1 Extraction of probability of events from traces

A trace can be represented as a string, where each trace record is encoded by a unique character. There exists a number of ways to encode the character. We concentrate on the following three character types c:

1. Record's function name (F),
2. Record's type (FT),
3. Record's function names, type, and depth in the call tree (FTD).

In addition, we can generate consecutive and overlapping substrings[18] of length l from a string. We call such substrings l-words. For example, a string “ABCA” contains the following 2-words: “AB”, “BC”, and “CA”.

One can think of a trace as a message generated by a source with source dictionary $A = \{a_1, a_2, \ldots, a_n\}$ consisting of n l-words a_i, and discrete probability distribution $P = \{p_1, p_2, \ldots, p_n\}$, where p_i is probability of a_i. The dictionaries A and their respective probability distributions P for various values of c and l for the trace given in are shown in Table 4.

---

[18] The substring can start at any character i, where $i \le n - l + 1$.

Table 4. Dictionaries of a trace given in Figure 11

C	l	n	A	P
F	1	2	f1, f2	1/3, 2/3
F	2	3	f1-f2, f2-f2, f2-f1	1/5, 3/5, 1/5
F	3	3	f1-f2-f2, f2-f2-f2, f2-f2-f1	1/4, 1/2, ¼
FT	1	4	f1-entry, f1-exit, f2-entry, f2-exit	1/6, 1/6, 1/3, 1/3
FTD	1	6	f1-entry-depth1, f1-exit-depth1, f2-entry-depth2, f2-exit-depth2, f2-entry-depth3, f2-exit-depth3	1/6, 1/6, 1/6, 1/6, 1/6, 1/6,

Let us define a function α that, given a trace t, will return a discrete probability distribution P for l-words of length l and characters of type c:

$$P \leftarrow \alpha(t; l, c). \tag{3.1}$$

The above empirical probability distribution P can now be used to calculate the entropy of a given trace for a specific l-word with characters of type c. We suppress the dependence of P (and the individual p_is) on t, l, and c. Let us now define entropies and discuss how we can utilize P in calculation of these entropies.

3.2.2 Entropies and traces

The Shannon entropy [21] is defined as

$$H_S(P) = -\sum_{i=1}^{n} p_i \log_b p_i, \tag{3.2}$$

where P is the vector containing probabilities of the n states, and p_i is the probability of i-th state. Logarithm base b controls the units of entropy. In this paper we set $b = 2$, measuring entropy in bits.

Three extended entropies, Landsberg-Vedral [12], Rényi [19], and Tsallis [22] are defined, respectively as:

$$H_L(P;q) = \frac{1-1/Q(P;q)}{1-q},$$
$$H_R(P;q) = \frac{\log_2\left[Q(P;q)\right]}{1-q}, \text{ and} \quad (3.3)$$
$$H_T(P;q) = \frac{Q(P;q)-1}{1-q},$$

where $q \geq 0$ is the entropy index and

$$Q(P;q) = \sum_{i=1}^{n} p_i^q. \quad (3.4)$$

The extended entropies reduce to the Shannon entropy (by L'Hôpital's rule) when $q = 1$. The extended entropies are more sensitive to states with small probability of occurrence than the Shannon entropy for $0 < q < 1$. Setting $q > 1$ leads to increased sensitivity of the extended entropies to states with high probability of occurrence.

The entropy, Z, of a trace, t, for a given l, c, and q is calculated by inserting the output of Equation (3.1) into one of the entropies described in Equations (3.2) and (3.3):

$$Z \leftarrow H_E\left[\alpha(t;l,c);q\right] \quad (3.5)$$

where $E \in \{L, R, T, S\}$. Note that if $E = S$, this is the Shannon entropy and q is ignored.

3.3 Usage of entropies for classification of traces

A typical scenario for trace comparison is the following. A software service analyst receives a phone call from a customer reporting software failure. The analyst needs to quickly determine the root cause of this failure and identify if 1) this is a rediscovery of a known defect exposed by some other customer in the past or 2) this is a newly discovered defect. If the first hypothesis is correct, then the analyst will be able to quickly provide the customer with a fix or describe a workaround for the problem. If the second hypothesis is correct the analyst must alert the maintenance team and start a full scale

investigation to identify the root-cause of this new problem. In each case, time is of the essence -- the faster the root cause is identified, the faster the customer will receive a fix to the problem and become less unsatisfied.

In order to validate the first hypothesis, the analyst asks the customer to reproduce the problem with a trace capturing facility enabled. The analyst can then compare the newly collected trace against a library of existing traces collected in the past (with known root-causes of the problems) and identify potential candidates for rediscovery. To identify a set of traces related to similar functionality the library traces are usually filtered by names of functions present in the trace of interest. After that the filtered subset of the library traces is examined manually to identify common patterns with the trace of interest.

If the analyst finds an existing trace with common patterns then the first hypothesis holds. Otherwise the analyst concludes[19] that this failure relates to a newly discovered defect and that the second hypothesis is valid. With tens of thousands of traces in the library the manual approach becomes laborious. This process is similar in nature to usage of an Internet search engine. A user provides to the search engine keywords of interest and the engine's algorithm returns a list of web pages ranked according to their relevance to keywords. The user examines the returned pages to identify pages most relevant to her.

To automate this approach using entropies as fingerprints, we need an algorithm that would compare a trace against a set of traces, rank this set based on the relevance to a trace of interest, and then return the top X closest traces for manual examination to the analyst. In order to implement this algorithm, we need a measure of distance between a pair of traces to quantify their closeness described in Section 3.3.1, and the ranking algorithm described in Section 3.3.2. Efficiency of the algorithm is analyzed in

---

[19] This is a simplified description of the analysis process. In practice the analyst will examine defects with similar symptoms, consult with her peers, search a database with descriptions of existing problems, etc.

Section 3.3.3. A drawback associated with usage of entropies as fingerprints is shown in Section 3.3.4.

3.3.1 Measure of distance between a pair of traces

We can obtain multiple entropy-based fingerprints for a trace by varying values of E, q, l and c. Let us denote a complete set of 4-tuples of $[E,q,l,c]$ as M. We define the distance between a pair of traces t_i and t_j as:

$$D(t_i,t_j;M)=\sqrt{\sum_{k=1}^{m}\left\{\frac{H_{E_k}\left[\alpha(t_i;l_k,c_k);q_k\right]-H_{E_k}\left[\alpha(t_j;l_k,c_k);q_k\right]}{\max\left\{H_{E_k}\left[\alpha(t;l_k,c_k);q_k\right]\right\}}\right\}^2}, \quad (3.6)$$

where m is the number of elements in M, and $\max\left\{H_{E_k}\left[\alpha(t;l_k,c_k);q_k\right]\right\}$ denotes the maximum value of H_{E_k} for the complete set of traces under study for a given q_k, l_k, and c_k. This denominator is used as a normalization factor to set equal weights to fingerprints related to different 4-tuples in M.

Formally (3.6), satisfies three of the four usual conditions of a metric:

$$\begin{aligned}&D(t_i,t_j;M)\geq 0,\\&D(t_i,t_j;M)=D(t_j,t_i;M),\\&D(t_i,t_k;M)\leq D(t_i,t_k;M)+D(t_k,t_j;M).\end{aligned} \quad (3.7)$$

However, the fourth condition $D(t_i,t_j;M)=0 \Leftrightarrow t_i=t_j$ (identity of indiscernibles) holds true only for the fingerprints of traces; the actual traces may be different even if their entropies are the same. In other words, the identity of indiscernibles axiom only “half” holds: $t_i=t_j \Rightarrow D(t_i,t_j;M)=0,$ but $D(t_i,t_j;M)=0 \not\Rightarrow t_i=t_j.$ As such, D represents a “pseudo-metric”. Note that $D(t_i,t_j;M)\in[0,\infty)$ and our hypothesis is the following: the smaller the value of D, the closer the traces.

Note that for a single pair of entropy-based fingerprints the normalization factor can be omitted and we define D as

$$D(t_i,t_j;E,q,l,c) = \left| H_E\left[\alpha(t_i;l,c);q\right] - H_E\left[\alpha(t_j;l,c);q\right] \right|. \qquad (3.8)$$

We now define an algorithm for ranking a set of traces with respect to the trace of interest.

3.3.2 Trace-ranking algorithm

Given a task of identifying top X closest classes of traces from a set of traces, T, closest to trace t we resort to the following pseudo-algorithm:

1. Calculate distances between t and each trace in T;

2. Order traces in T by their distance to trace t in ascending order;

3. Replace the vector of sorted traces with the vector of classes (e.g., defect IDs) to which these traces map;

4. Keep the first occurrence (i.e., the closest trace) of each class in the vector and remove the rest;

5. Calculate the ranking of classes taking into account ties using the "modified competition ranking"[20] approach;

6. Return a list of classes with ranking smaller than or equal to X.

---

[20] The "modified competition ranking" assigns the same rank to items compared equal and leaves the gap before the set of the items with the same rank. For example, if A is ranked ahead of B and C (considered equal), which in turn are ranked ahead of D then the ranks will be performed a follows: A gets rank 1, B and C gets rank 3, and D gets rank 4.

The “modified competition ranking” can be interpreted as a worst-case-scenario approach. The ordering of traces of equal ranks is arbitrary; therefore we are looking at the case when the most relevant trace will always reside at the bottom of the returned list. To be conservative, we consider the outcome in which our method returns a trace in the top X positions as being in the X-th position.

Now consider an example of the algorithm:

3.3.2.1 Traces ranking algorithm: example

Suppose that we have five traces t_i, $i = 1..5$. The traces related to four software defects d_j, $j = 1..4$ as shown in Table 5.

Table 5. Example: Relation between traces and defects.

Defect	Trace
d_1	t_5
d_2	t_1, t_3
d_3	t_4
d_4	t_2

Suppose that we calculate distances between traces using some measure of distance. The distances between trace t and $t_{1..5}$ and the defects' ranks obtained using these hypothetical calculations are given in Table 6. The traces are ranked based on the modified competition ranking schema. Trace t_2 is the closest to t, hence d_4 (to which t_2 is related) gets ranking number 1. Traces t_1 and t_4 have the same distance to t, therefore, d_2 and d_3 get the same rank. Based on the ranking schema algorithm we leave a gap before the set of items with the same rank and assign rank 3 to both classes. Traces t_3 and t_5 also have the same distance to t; however t_3 should be ignored since it relates to the already ranked defect d_2. This leads to assigning rank 4 to d_1. The resulting sets of top X traces for different values of X are shown in Table 7.

Table 6. Example: Traces sorted by distance and ranked

t_i	Distance between t and t_i	Class (defect ID) of trace t_i	Rank
t_2	0	d_4	1
t_1	7	d_2	3
t_4	7	d_3	3
t_3	9	d_2	--
t_5	9	d_1	4

Table 7. Example: Top 1-4 defects

Top X	Set of defects in Top X
Top 1	d_4
Top 2	d_4
Top 3	d_4, d_2, d_3
Top 4	d_4, d_2, d_3, d_1

3.3.3 Traces ranking algorithm: efficiency

The number of operations C needed by the ranking algorithm is given by

$$\begin{aligned} C = & \underbrace{c_1 O(|M||T|)}_{\text{Step1}} + \underbrace{c_2 O(|T|\log|T|)}_{\text{Step2}} + \underbrace{c_3 O(|T|)}_{\text{Step3}} \\ & + \underbrace{c_4 O(|T|)}_{\text{Step4}} + \underbrace{c_5 O(|T|)}_{\text{Step5}} + \underbrace{c_6 O(1)}_{\text{Step6}} \\ & \overset{\substack{|T|\to\infty, \\ c_1 \gg \{c_3, c_4, c_5\}}}{\approx} \underbrace{c_1 O(|M||T|)}_{\text{Step1}} + \underbrace{c_2 O(|T|\log|T|)}_{\text{Step2}}, \end{aligned} \tag{3.9}$$

where c_i is a constant number of operations associated with i-th step, and $|\cdot|$ represents the number of elements in a given set. The coefficients c_3, c_4 and c_5 are of much smaller order than c_1 and hence terms corresponding to Steps 3, 4 and 5 do not contribute significantly to C. Pair-wise distance calculation, using (3.6), requires $O(|M|)$ operations. Therefore, calculation of distances between traces (Step 1) requires $O(|M||T|)$ operations. Assuming that $|M|$ remains constant, the number of operations grows linearly with $|T|$. The average sorting algorithm, required by Step 2 (sorting of

traces by their distance to trace t), needs $O(|T|\log|T|)$ operations [2]. Usually, $c_1 \gg c_2$; this implies that a user may expect to see linear relation between C and $|T|$ (even for large $|T|$), in spite of the loglinear complexity of the second term in (3.9).

The amount of storage needed for entropy-based fingerprints data (used by (3.6)) is proportional to

$$\underbrace{\phi|M||T|}_{a}+\underbrace{\phi|M|}_{b}=\phi|M|(|T|+1), \tag{3.10}$$

where ϕ is the number of bytes needed to store a single fingerprint value. Term a is the amount of storage needed for entropy-based fingerprints for all traces in T, and term b is the amount of storage needed for the values of $\max\left\{H_{E_k}\left[\alpha(t;l_k,c_k);q_k\right]\right\}$ from (3.6). Assuming that $|M|$ remains constant, the data size grows linearly with $|T|$.

3.3.4 Entropies as fingerprints: drawback

The drawback associated with entropies comes from the fact that entropies cannot differentiate dictionaries of events, since entropy formulas operate only with probabilities of events. Therefore, entropies of strings “f1-f2-f3-f1” and “f4-f5-f6-f4” will be exactly the same for any value of E, l, c, and q. The simplest solution is to do a pre-filtering of traces in T in the spirit of the SIFT framework described in Section 3.1. For example, one can filter out all the traces that do not contain “characters” (e.g., function names) present in the trace of interest before using entropy-based fingerprints.

3.4 Validation case study

We hypothesize that predictive classification power will vary with change in E, l, c, and q. In order to study the classification power of $H_E\left[\alpha(t;l,c);q\right]$ we will analyze Cartesian products of the following sets of variables:

1. $E \in (S,L,R,T)$,

2. $l \in (1,2,\ldots,7)$,

3. $q \in (0,10^{-5},10^{-4},\ldots,10^{1},10^{2})$,

4. $c \in (F, FT, FTD)$.

Let us denote the complete set of parameters obtained by the Cartesian product as Λ.

Our software under study, called the Siemens suite, was first developed by Hutchins et al. [10] at the Siemens Corporate Research. It was further augmented and publicly made available at Software-artifact Infrastructure Repository [27, 26]. This software suite has been used by a large number of studies on defect analysis in the last decade (see [11, 17] for literature review).

The Siemens suite [10] contains seven programs. Each program has one original version and a number of faulty versions. A faulty version is a variant of the original version by one fault. A fault (changed source code from the original version) was seeded manually by Hutchins et al. [10]. A fault can span over multiple lines of source code and multiple functions. Each program comes with a collection of test cases, applicable to all faulty versions and the original program. A fault can be identified if the output of a test case on the original version differs from the output of the same test case on a faulty version of the program.

In this study, we experimented with the largest program "Replace" of the Siemens suite. It has 517 lines of code, 21 functions, 31 different faulty versions. There were 5542 test cases shared across all the versions. Out of these 31×5542 test cases, 4266 ($\approx 2.5\%$ of the total number of test cases) caused a program failure when exposed to the faulty program, i.e., were able to catch a defect. The remaining test cases were probably unrelated to the 31 defects. The traces for failed test cases were collected using a tool called Etrace [25]. The tool captures sequences of function calls for a particular software

execution such as the one shown in Figure 11. In other words, we collected 4266 function-call level failed traces for 31 faults (faulty versions) of the “Replace” program[21].

The distribution of the number of traces mapped to a particular defect (version) is given in Figure 12. Descriptive statistics of trace length are given in Table 8. The length ranges between 11 and 101400 records per trace; average length is 623 records per trace. Average dictionary sizes for various values of c are given in Figure 13. Note that as l gets larger, the dictionary sizes for all c start to converge.

Table 8. Descriptive statistics of length of traces

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
11	218	380	623.3	678	101400

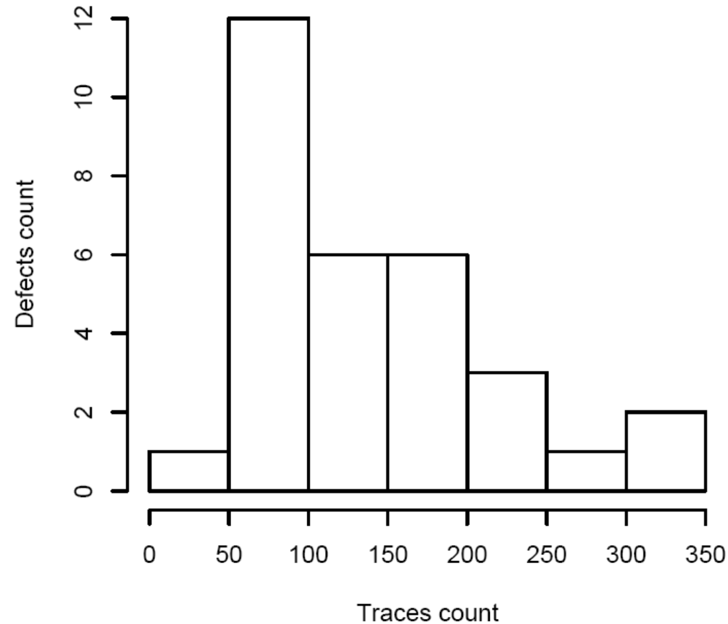

Figure 12. Distribution of the number of traces per defect (version)

[21] The “Replace” program had 32 faults, but the tool “Etrace” was unable to capture the traces of segmentation fault in one of the faulty versions of the “Replace” program. This problem was reported also by other researchers [11].

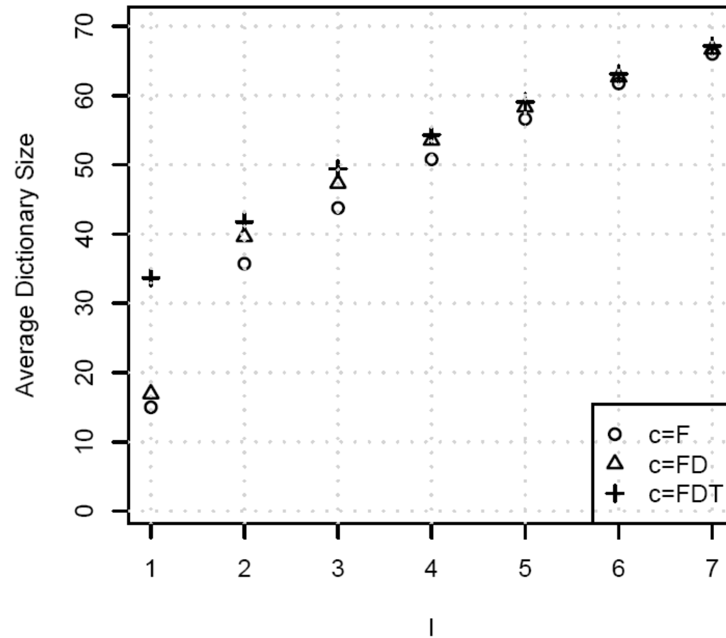

Figure 13. Dictionary size for various values of l and c

Each of the traces contains at least one shared function. Therefore, we skip the pre-filtering step. Note that direct comparison with existing trace comparison techniques is not possible since 1) the authors focus on identification of faulty functions [11, 17] instead of identification of defect IDs and 2) the authors [11] analyze a complete set of programs in the Siemens suite while we focus only on one program (Replace).

The case study is split into three parts: 1) analysis of the individual classification power of each $H_E\left[\alpha(t;l,c);q\right]$ in Section 3.4.1; 2) analysis of the classification power of the complete set of entropies in Section 3.4.2.

3.4.1 Analysis of individual entropies

Analysis of the classification power of individual entropies is performed using 10-fold cross-validation. The validation process is designed as follows:

1. Randomly partition 4266 traces into 10 bins

2. For each set of parameters E, l, c, q

    a. For each bin

        i. Tag traces in a given bin as a validating set of data and traces in the remaining nine bins as a training set

        ii. For each trace t in the validating set calculate the rank of t's class (defect ID) in the training set using the algorithm in Section 3.3.2[22] with (3.8) as the measure of distance and with the set of parameters E, l, c, q

    b. Average information about ranks of the "true" classes and store this data for further analysis

Our findings show that the best results are obtained for H with $E \in (L, R, T)$, $l = 3$, $q \in (10^{-5}, 10^{-4})$, and $c = FDT$. Based on 10-fold cross validation, the entropies with these parameters were able to correctly classify $\approx 21.6\% \pm 1.1\%$ [23] of Top 1 defects and $\approx 57.6\% \pm 1.5\%$ of Top 5 defects (see Table 9 and Figure 14). Based on the standard deviation data in Table 9, all six entropies show robust results. However, the results become slightly more volatile for high ranks (see Figure 15). Let us analyze these findings in details.

---

[22] Technically, in order to identify the true ranking one needs to tweak Step 6 of the algorithm and return a vector of 2-tuples [class, rank].

[23] 95% confidence interval, calculated as $\pm q(0.975, 9) / \sqrt{10} \times \text{standard deviation}$, where $q(p, df)$ represents quantile function of the *t*-distribution, where *p* is the probability and *df* is the degrees of freedom.

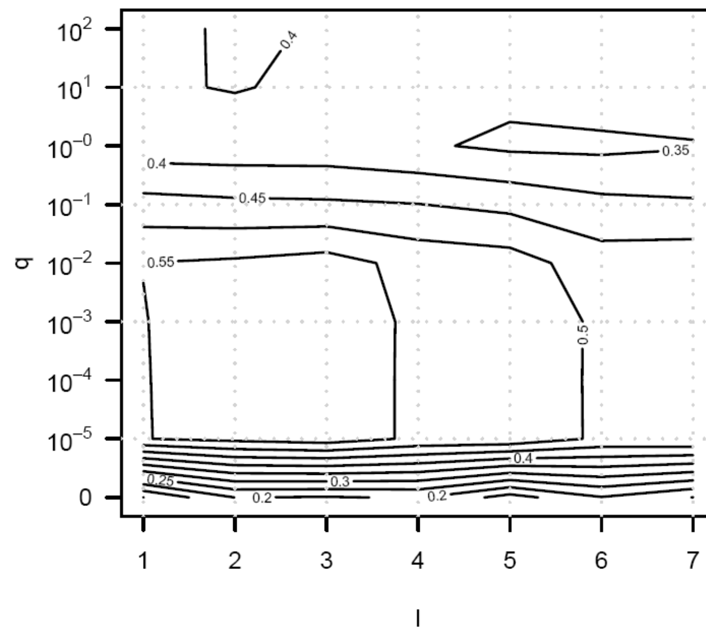

Figure 14. Interpolated average fractions of correctly classified traces in Top 5 (based on 10-fold cross validation) for $E = L$ and $c = FDT$. for different values of l and q.

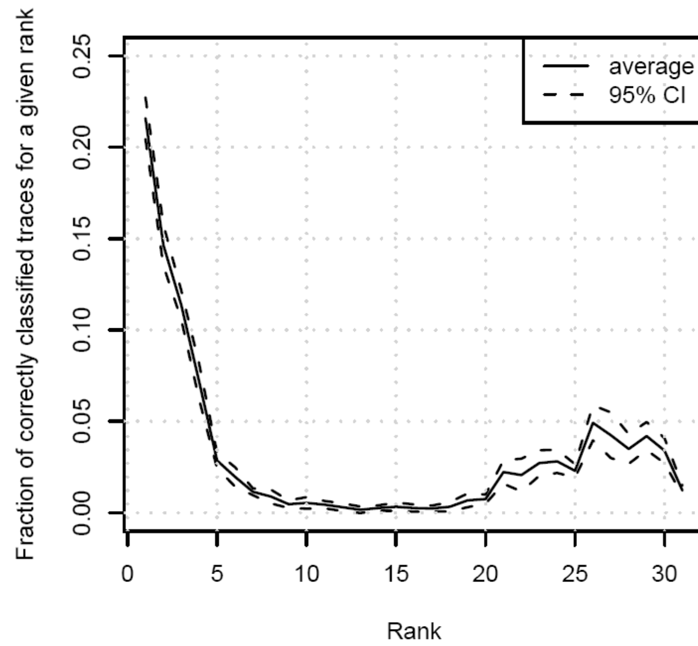

Figure 15. Fraction of correctly classified traces in Top 5 for $E = L$, $l = 3$, $q = 10^{-5}$, and $c = FDT$. Solid line shows the average fraction of correctly classified traces in 10 folds; dotted line shows pointwise 95% confidence interval (95% CI) of the average.

Table 9. Fraction of correctly classified traces in Top X for 1) $H_E\left[\alpha(t;l,c);q\right]$ with $E \in (L,R,T)$, $q \in (10^{-5},10^{-4})$, $l=3$, and $c = FDT$, and 2) set of entropies Λ; based on 10-fold cross validation. Average fraction of correctly classified traces in 10 folds is denoted by "Avg."; plus-minus 95% confidence interval denoted by "95% CI".

Top X	$E=L$				$E=R$				$E=T$				Λ	
	$q=10^{-4}$		$q=10^{-5}$		$q=10^{-4}$		$q=10^{-5}$		$q=10^{-4}$		$q=10^{-5}$			
	Avg.	95%CI	Avg.	95%CI	Avg.	95%CI	Avg.	95%CI	Avg.	95%CI	Avg.	95%CI	Avg.	95%CI
1	0.2159	0.0114	0.2159	0.0114	0.2159	0.0114	0.2159	0.0114	0.2159	0.0114	0.2159	0.0114	0.2972	0.0151
2	0.3624	0.0180	0.3624	0.0180	0.3624	0.0180	0.3624	0.0180	0.3621	0.0180	0.3621	0.0180	0.4643	0.0189
3	0.4761	0.0219	0.4761	0.0219	0.4761	0.0219	0.4761	0.0219	0.4761	0.0219	0.4761	0.0219	0.5790	0.0181
4	0.5478	0.0164	0.5478	0.0164	0.5478	0.0164	0.5478	0.0164	0.5478	0.0165	0.5478	0.0165	0.6017	0.0165
5	0.5764	0.0149	0.5764	0.0149	0.5764	0.0149	0.5764	0.0149	0.5766	0.0151	0.5766	0.0151	0.6153	0.0170
6	0.5961	0.0174	0.5961	0.0174	0.5961	0.0174	0.5961	0.0174	0.5956	0.0177	0.5956	0.0177	0.6247	0.0179
7	0.6076	0.0174	0.6076	0.0174	0.6076	0.0174	0.6076	0.0174	0.6076	0.0173	0.6076	0.0173	0.6322	0.0171
8	0.6165	0.0166	0.6165	0.0166	0.6165	0.0166	0.6165	0.0166	0.6169	0.0165	0.6169	0.0165	0.6385	0.0170
9	0.6212	0.0162	0.6212	0.0162	0.6212	0.0162	0.6212	0.0162	0.6228	0.0158	0.6228	0.0158	0.6416	0.0171
10	0.6266	0.0167	0.6266	0.0167	0.6266	0.0167	0.6266	0.0167	0.6280	0.0163	0.6280	0.0162	0.6453	0.0173
11	0.6310	0.0172	0.6310	0.0172	0.6310	0.0172	0.6310	0.0172	0.6331	0.0167	0.6331	0.0168	0.6472	0.0164
12	0.6341	0.0177	0.6341	0.0177	0.6341	0.0177	0.6341	0.0177	0.6357	0.0175	0.6357	0.0175	0.6488	0.0164
13	0.6357	0.0170	0.6357	0.0170	0.6357	0.0170	0.6357	0.0170	0.6381	0.0169	0.6381	0.0170	0.6495	0.0165
14	0.6383	0.0173	0.6383	0.0173	0.6383	0.0173	0.6383	0.0173	0.6406	0.0170	0.6406	0.0169	0.6505	0.0161
15	0.6416	0.0164	0.6416	0.0164	0.6420	0.0162	0.6416	0.0164	0.6432	0.0162	0.6432	0.0162	0.6512	0.0160
16	0.6441	0.0167	0.6441	0.0167	0.6441	0.0167	0.6441	0.0167	0.6453	0.0166	0.6453	0.0166	0.6517	0.0159
17	0.6463	0.0157	0.6465	0.0157	0.6463	0.0157	0.6465	0.0157	0.6474	0.0156	0.6477	0.0155	0.6528	0.0157
18	0.6495	0.0147	0.6498	0.0146	0.6495	0.0147	0.6498	0.0146	0.6509	0.0144	0.6512	0.0144	0.6552	0.0152
19	0.6563	0.0150	0.6566	0.0149	0.6563	0.0150	0.6566	0.0149	0.6577	0.0147	0.6580	0.0147	0.6580	0.0149
20	0.6641	0.0163	0.6641	0.0163	0.6641	0.0163	0.6641	0.0163	0.6655	0.0161	0.6655	0.0160	0.6636	0.0161
21	0.6863	0.0142	0.6863	0.0142	0.6863	0.0142	0.6863	0.0142	0.6873	0.0141	0.6873	0.0142	0.6730	0.0167
22	0.7070	0.0106	0.7070	0.0106	0.7070	0.0106	0.7070	0.0106	0.7079	0.0104	0.7079	0.0104	0.7004	0.0128
23	0.7342	0.0117	0.7342	0.0117	0.7339	0.0117	0.7339	0.0117	0.7342	0.0114	0.7342	0.0112	0.7225	0.0138
24	0.7623	0.0092	0.7623	0.0092	0.7623	0.0092	0.7623	0.0092	0.7628	0.0089	0.7628	0.0088	0.7482	0.0173
25	0.7853	0.0076	0.7853	0.0076	0.7855	0.0074	0.7855	0.0074	0.7865	0.0072	0.7865	0.0072	0.7771	0.0121
26	0.8345	0.0082	0.8345	0.0082	0.8347	0.0077	0.8347	0.0077	0.8352	0.0082	0.8352	0.0082	0.8190	0.0091
27	0.8769	0.0097	0.8769	0.0097	0.8772	0.0097	0.8772	0.0097	0.8776	0.0097	0.8776	0.0097	0.8687	0.0088
28	0.9119	0.0082	0.9119	0.0082	0.9119	0.0082	0.9119	0.0082	0.9142	0.0083	0.9142	0.0083	0.9243	0.0094
29	0.9538	0.0054	0.9538	0.0054	0.9538	0.0054	0.9538	0.0054	0.9550	0.0054	0.9550	0.0059	0.9655	0.0042
30	0.9878	0.0027	0.9878	0.0027	0.9878	0.0027	0.9878	0.0027	0.9887	0.0027	0.9887	0.0027	0.9977	0.0016
31	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000

The l-words with $l = 3$ provide the best results based on the fraction of correctly classified traces in Top 5 (see Figure 16), suggesting that chains of three events provide optimal balance between the amount of information in a given l-word and the total number of words. As l gets larger, the amount of data becomes insufficient to get a good estimate of the probabilities.

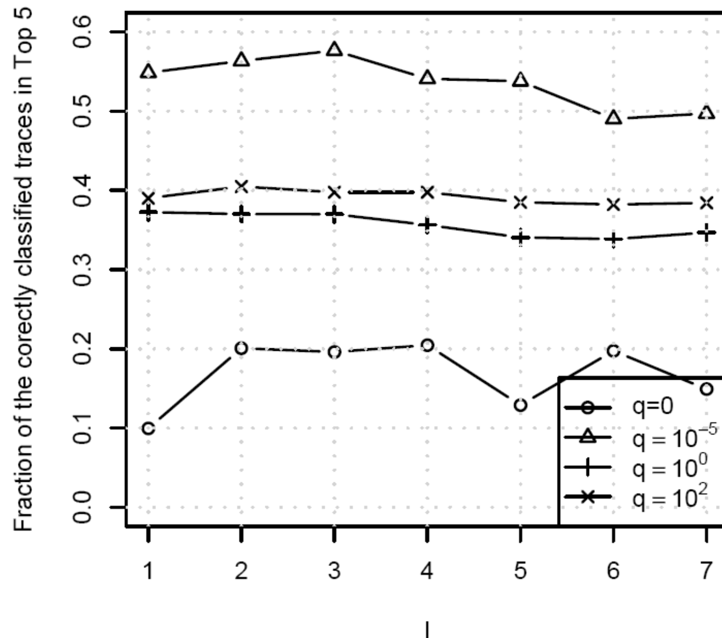

Figure 16. Average fraction of correctly classified traces in Top 5 for various values of l; $E = L$, $q \in (0, 10^{-5}, 10^{0}, 10^{2})$, $c = FDT$

Comparison of the average fraction of correctly classified traces in Top 5 for the three values of c shows that FDT outperforms FD and F (see Table 10). However, the difference between three values is marginal: for example 57.6% in Top 5 for $c = FDT$ vs. 56.2% for $c = F$. The fact that FDT outperforms the remaining character types is expected, since FDT contains the largest amount of information. However, the addition of information about type of trace point (entry or exit) does not significantly contribute to the classification power of the algorithm. Note that even though more time is needed to

calculate the FDT-based entropies (since the dictionary of FDTs will be twice as large as the dictionary of FDs for small l), the comparison time remains the same (since the probabilities of l-words, P, map to a scalar value via the entropy function for all values of c).

Table 10. Percent of correctly classified traces in Top X for $H_E\left[\alpha(t;l,c);q\right]$, $E=L$, $l=3$, and $q=\{10^{-4},10^{-5}\}$

Top X	$c=F$	$c=FD$	$c=FDT$
Top 1	21.7%	20.9%	21.6%
Top 2	37.2%	35.3%	36.2%
Top 3	49.5%	46.7%	47.6%
Top 4	54.0%	53.5%	54.8%
Top 5	56.2%	56.6%	57.6%

Our findings show that the extended entropies outperform the Shannon entropy[24] for $q<1$ and $q>1$ (see Figure 17). However, performance of extended entropies with $q<1$ is significantly better than with $q>1$, suggesting that rare events are more important than frequent events for classification of defects in this dataset. The best results are obtained for $q=10^{-4}$ and $q=10^{-5}$.

It is interesting to note that classification performance is almost identical for H with $E\in(L,R,T)$, l=3, $q\in(10^{-5},10^{-4})$, and $c=FDT$. We believe that this fact can be explained as follows: the key contribution to the ordering of similar traces (with similar dictionaries) for entropies with $q\to 0$ is affected mainly by a function of probabilities of traces' events. This function is independent of E and q and depends only on l and c, see Appendix 3.6 for details.

[24] We do not explicitly mention entropy values on the figures. However, extended entropy values with $q=1$ correspond to values of the Shannon entropy.

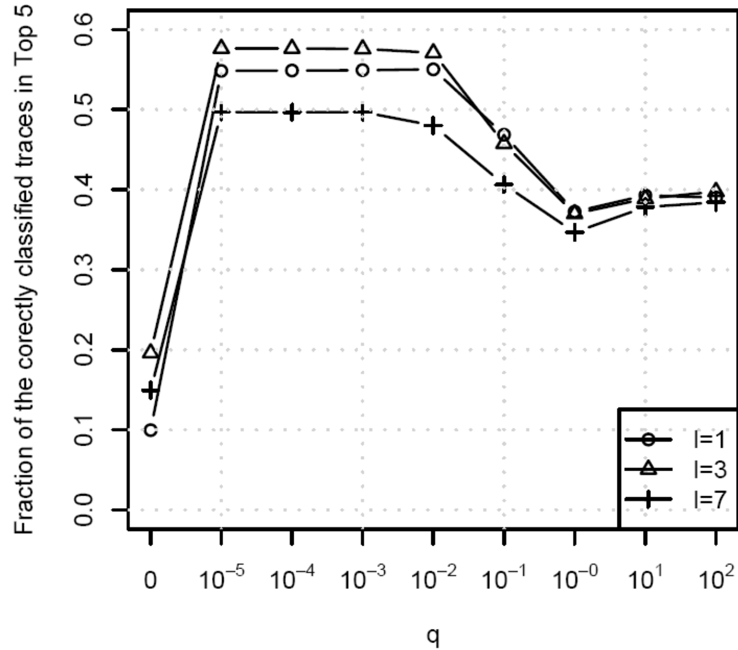

Figure 17. Average fraction of correctly classified traces in Top 5 for various values of q; $E = L$, $l \in (1,3,7)$, $c = FDT$

3.4.2 Analysis of the complete set of entropies

Analysis of the classification power for the complete set of entropies is performed using 10-fold cross-validation in a similar manner to the process described in Section 3.4.1. However, instead of calculating distances for each H independently, we now calculate distances between traces by utilizing values of H for all parameter sets in Λ simultaneously. The validation process is designed as follows

1. Randomly partition 4266 traces into 10 bins

    a. For each bin

        i. Tag traces in a given bin as a validating set of data and traces in the remaining nine bins as a training set;

ii. For each trace t in the validating set calculate the rank of t's class (defect ID) in the training set using the algorithm in Section 3.3.2 with equation and all[25] the 4-tuples of parameters in Λ.

b. Average information about ranks of the "true" classes and store this data for further analysis.

The results shown in Table 9 show the increase of predictive power: in the case of Top 1 the results improved from 21.6% (for individual entropies) to 29.7% (for all entropies combined); for Top 5 from 57.6% to 61.5%. A significant increase in computational effort (the number of entropy fingerprints increases from 1 to 504) does not yield dramatic improvement: the 7% increase in power for predicting Top 5 matches comes at a 503-fold increase in computational effort. We leave the resulting balance between cost and benefit for each individual analyst to make.

3.5 Summary

In this work we analyze the applicability of entropies to predictive classification of traces related to software defects. Our validating case study shows promising performance of extended entropies with emphasis on rare events $\left(q \in \left\{10^{-5}, 10^{-4}\right\}\right)$. The events are based on triplets (3-words) of "characters" incorporating information about function name, depth of function call, and type of probe point ($c = FDT$).

In the future, we are planning to increase the number of datasets under study, derive additional measures of distance (e.g., using tree classification algorithms) and identify an optimal set of combinations of parameters.

---

[25] We had to exclude a subset of entropies with $E = L$, $q = 10^2$ for all l and c from Λ. The values of entropies obtained with these parameters are very large ($> 10^{100}$), which leads to numeric instability of (6). We keep just one of the various named $q = 1$ entropies to avoid redundancy.

References

[1] J. Aczél and Z. Daróczy. *On measures of information and their characterizations*. Academic Press, 1975.

[2] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009.

[3] Domenico Cotroneo and Roberto Pietrantuono and Leonardo Mariani and Fabrizio Pastore. Investigation of failure causes in workload-driven reliability testing. *Proc. of the 4th international wokshop on software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 78--85, 2007.

[4] Matt Davison and J. S. Shiner. Extended Entropies And Disorder. *Advances in Complex Systems (ACS)*, 8(01):125--158, 2005.

[5] Matthew Davison and Mechelle Sophia Gittens and David Richard Godwin and Nazim H. Madhavji and Andriy Vladimir Miranskyy and Mark Francis Wilding. Computer software test coverage analysis. 2006.

[6] W. Ebeling and G. Nicolis. Word frequency and entropy of symbolic sequences: a dynamical perspective. *Chaos, Solitons and Fractals*, 2(6):635--650, 1992.

[7] Sebastian Elbaum and Satya Kanduri and Anneliese Andrews. Trace anomalies as precursors of field failures: an empirical study. *Empirical Softw. Eng.*, 12(5):447--469, 2007.

[8] Sebastian Elbaum and Gregg Rothermel and Satya Kanduri and Alexey G. Malishevsky. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Control*, 12(3):185--210, 2004.

[9] Murali Haran and Alan Karr and Alessandro Orso and Adam Porter and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 146--155, 2005.

[10] Monica Hutchins and Herb Foster and Tarak Goradia and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering*, pages 191--200, 1994.

[11] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273--282, 2005.

[12] Peter T. Landsberg and Vlatko Vedral. Distributions and channel capacities in generalized statistical mechanics. *Physics Letters A*, 247(3):211--217, 1998.

[13] Wenke Lee and Salvatore J. Stolfo and Philip K. Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. *In AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, :50--56, 1997.

[14] Leonardo Mariani and Mauro Pezzé. Dynamic Detection of COTS Component Incompatibility. *IEEE Software*, 24(5):76--85, 2007.

[15] A. V. Miranskyy and M. S. Gittens and N. H. Madhavji and C. A. Taylor. Usage of Long Execution Sequences for Test Case Prioritization. *Supplemental Proceedings of 18th IEEE International Symposium on Software Reliability Engineering*, 2007.

[16] A. V. Miranskyy and N. H. Madhavji and M. S. Gittens and M. Davison and M. Wilding and D. Godwin and C. A. Taylor. SIFT: a scalable iterative-unfolding technique for filtering execution traces. *Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 274--288, 2008.

[17] S. S. Murtaza and M. Gittens and Z. Li and N. H. Madhavji. F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field. *Proceedings of the 2010 conference of the center for advanced studies on collaborative research*, 2010, to appear.

[18] Andy Podgurski and David Leon and Patrick Francis and Wes Masri and Melinda Minch and Jiayang Sun and Bin Wang. Automated support for classifying software failure reports. *Proceedings of the 25th International Conference on Software Engineering*, pages 465--475, 2003.

[19] Alfréd Rényi. *Probability theory*. North-Holland Pub. Co., 1970.

[20] Gregg Rothermel and Mary Jean Harrold and Jeffery Ostrin and Christie Hong. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. *Proceedings of the International Conference on Software Maintenance*, pages 34, 1998.

[21] Claude E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:623--656, 1948.

[22] Constantino Tsallis. Possible generalization of Boltzmann-Gibbs statistics. *Journal of Statistical Physics*, 52(1):479--487, 1988.

[23] Susana Vinga and Jonas S Almeida. Rényi continuous entropy of DNA sequences. *Journal of Theoretical Biology*, 231(3):377--388, 2004.

[24] Chun Yuan and Ni Lao and Ji-Rong Wen and Jiwei Li and Zheng Zhang and Yi-Min Wang and Wei-Ying Ma. Automated known problem diagnosis with event traces. *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 375--388, 2006.

[25] Etrace. http://ndevilla.free.fr/etrace/, Accessed: November 15, 2010.

[26] Siemens Suite. http://pleuma.cc.gatech.edu /aristotle/Tools/subjects/, Accessed: November 15, 2010.

[27] Hyunsook Do and Sebastian Elbaum and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405--435, 2005.

[28] Hamou-Lhadj, Abdelwahab. Measuring the Complexity of Traces Using Shannon Entropy. *Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 489--494, 2008.

[29] Johan Moe and David A. Carr. Using execution trace data to improve distributed systems. *Software: Practice and Experience*, 32(9):889--906, 2002.

[30] R. Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99:89--112, 1997.

3.6 Appendix: Approximation of Equation (3.8)

We have observed that classification power of the $H_E\left[\alpha(t;l,c);q\right]$ is the highest when $q \to 0$. In order to explain this phenomenon let us expand $H_E\left[\alpha(t;l,c);q\right]$ using a Taylor series:

$$\begin{aligned}
H_L\left[\alpha(t_i;l,c);q\right] &\overset{q\to 0}{=} 1-\frac{1}{n_i}+q\left(\frac{A_i}{n_i^2}+\frac{n_i-1}{n_i}\right)+O(q^2),\\
H_R\left[\alpha(t_i;l,c);q\right] &\overset{q\to 0}{=} \log_2(n_i)+q\left[\frac{A_i}{n_i\log_e(2)}+\log_2(n_i)\right]+O(q^2), \qquad (3.11)\\
H_T\left[\alpha(t_i;l,c);q\right] &\overset{q\to 0}{=} n_i-1+q\left(A_i+n-1\right)+O(q^2),
\end{aligned}$$

where $A_i=\sum_{k=1}^{n_i}\log_e(p_k)$. By plugging (3.11) into (3.8) and assuming that for similar traces $n \approx n_i \approx n_j$, (3.8) becomes:

$$\begin{aligned} D(t_i, t_j; L, q, l, c) &\approx \frac{q}{n^2}\left|A_i - A_j\right|, \\ D(t_i, t_j; R, q, l, c) &\approx \frac{q}{n\log_e(2)}\left|A_i - A_j\right|, \\ D(t_i, t_j; T, q, l, c) &\approx q\left|A_i - A_j\right|, \end{aligned} \tag{3.12}$$

Equation (3.12) can be interpreted as follows. In the case when $q \to 0$ and dictionaries of a pair of traces are similar, the key contribution to the measure of distance is coming from the $\sum_{k=1}^{n_i} \log_e(p_k)$ term (which depends only on l and c) making the rest of the variables irrelevant (q and n become parts of scaling factors). This can be highlighted by solving a system of equations to identify conditions that generate the same ordering for three traces t_i, t_j, t_k for all extended entropies (using approximations from (3.12)):

$$\begin{cases} \frac{q}{n^2}\left|A_i - A_j\right| \leqslant \frac{q}{n^2}\left|A_i - A_k\right| \\ \frac{q}{n\log_e(2)}\left|A_i - A_j\right| \leqslant \frac{q}{n\log_e(2)}\left|A_i - A_k\right| \\ q\left|A_i - A_j\right| \leqslant q\left|A_i - A_k\right| \end{cases} \Rightarrow \left|A_i - A_j\right| \leqslant \left|A_i - A_k\right|. \tag{3.13}$$

In information theory $\log_e(p_k)$ is the “surprise” in receiving the bit *k* which occurs with probability p_k. Thus $\sum_{k=1}^{n_i} p_k \log_e(p_k)$ is the expected surprise or information (Shannon entropy). What about just $\sum_{k=1}^{n_i} \log_e(p_k)$? It scales with the total number of bits needed to specify each symbol. This is related to the problem of simulating processes in the presence of rare events, see [30] for details.

Chapter 4

4 Metrics of Risk Associated with Defects Rediscovery

Software defects rediscovered by a large number of customers affect various stakeholders and may: 1) hint at gaps in a software manufacturer's Quality Assurance (QA) processes, 2) lead to an overload of a software manufacturer's support and maintenance teams, and 3) consume customers' resources, leading to a loss of reputation and a decrease in sales.

Quantifying risks associated with the rediscovery of defects can help all of these stakeholders. In this chapter we present a set of metrics needed to quantify the risks. The metrics are designed to help: 1) the QA team to assess their processes; 2) the support and maintenance teams to allocate their resources; and 3) the customers to assess the risks associated with the use of the software product. The paper includes a validation case study showing application of these risk metrics to industrial data. To calculate the metrics we use mathematical instruments like the heavy-tailed Kappa distribution and the G/M/k queuing model.

4.1 Introduction

During in-house testing of a software product, the Quality Assurance (QA) team attempts to remove defects injected during software development. It is impossible to remove all defects before shipping the software product. As customers use the product, they *discover* defects that "escaped" the QA team. Upon defect reporting the software provider's maintenance team prepares and makes available a fix. A discovered defect is sometimes *rediscovered* by another customer. This rediscovery could occur because another customer finds the defect before the fix is available or has not been installed. Defects relating to rarely used software features will be rediscovered infrequently. However, defects relating to popular and extensively used features may affect a significant percentage of customers.

Frequently rediscovered defects (affecting many customers) can cause an *avalanche* of requests, defined as a large number of requests for fix patches from multiple customers within a short timeframe. Because different software versions are run on different software and hardware platforms, each customer may require a different fix patch.

Avalanches have significant consequences. Support personnel will experience a heavy volume of support requests. The maintenance team will need to prepare a large number of special builds, while customers await the official fix. On the other side, the customers' system administrators will need to spend time assessing the fix's risk and distributing it to their systems. An inordinate number of defects may diminish the provider's reputation and result in decreased software sales.

Frequent rediscovery of a defect suggests that one or more common functionalities were not properly tested. Analysis of such defects is important to identify gaps in QA processes to prevent the future escape of similar defects.

Defect risk analysis is therefore important for software manufacturers and customers. We propose a set of quantitative risk metrics which can be used to assist:

- The support team's assessment of the potential number of repeated calls on the same subject, helping in personnel allocation;
- The maintenance team's estimation of the potential number of repeated special builds, assisting in resource allocation of team members;
- The QA team's assessment of trends in frequently rediscovered defects on release-to-release basis. If the trend shows increased defect rediscovery, QA processes must be improved. The resulting strategy to close testing process gaps can be derived by root cause analysis of frequently rediscovered defects;
- Customers assessment of risks associated with software product usage.

We present a validation case study showing applicability of these metrics to an industrial dataset of defects rediscovery. In order to model the data we derive a compound Kappa distribution and use the G/M/k queueing model.

Section 4.2 of this chapter reviews relevant work. Section 4.3 provides formal definitions and applications of the metrics. Section 4.4 provides a validation case study, showing application of the metrics to the industrial data. Finally, Section 4.5 concludes the chapter.

4.2 Related Research

The chapter's main contribution is a set of metrics for assessing defect rediscovery risks. The following metrics have been formulated by other authors: the number of rediscoveries per defect [1], the time interval between first and last rediscovery of a given defect[1] and the probability that a customer will observe failure in a given timeframe [2]. Our metrics are complementary to these three.

Our metrics can help in resource allocation of service and maintenance teams; these metrics rely on information about arrival of defect rediscoveries. Other authors have used counting processes [3] and regression models to help estimate staffing needs. However, the authors do not assess risks associated with under-staffing; hence our work complements theirs.

We use a G/M/k queue analysis to estimate staffing needs for delivery of special builds fixing rediscoveries for customers. Queuing theory tools have not yet been applied to this problem, although the load on a k-member service team delivering fixes for initial rediscovery of defects was modeled in [4] using *k* M/M/1 processes. Work has also been done on modeling the initial discovery repair time distribution [5] and predicting defect repair time based on attributes of past defect reports [6].

The second contribution of this paper is the introduction of a compound Kappa distribution, related to the family of heavy-tailed distributions, to model the data. While

previous work has observed that, depending on the dataset, distribution of defect rediscoveries is either thin-tailed (exponentially bounded) [7] or heavy-tailed [8],[9], many processes in software engineering are governed by heavy-tailed distributions [10]. Based on these observations, modeling the rediscovery distribution was performed using the empirical [8], geometric [7], lognormal [9], and Pareto [9] distributions. We found that none of these parametric models provided an adequate fit to our data. Therefore, we introduced a more flexible distribution, namely, the compound Kappa for the number of rediscoveries that also allows for tail-event information not available in the empirical distribution.

4.3 Metrics of Risk

Motivation for metric applications is described in Section 4.3.1 with their formal definitions deferred until Section 4.3.2.

4.3.1 Metrics Application

Metrics used by Support and Maintenance Teams, Quality Assurance Team and customers are given in Sections 4.3.1.1, 4.3.1.2 and 4.3.1.3, respectively.

4.3.1.1 Support and Maintenance Teams

Defect discovery related to common and frequently executed functionalities triggers a large number of support requests shortly after its initial discovery. This can be explained as follows:

Proactive requests for software fix: The software manufacturer publishes information about newly discovered defects on a regular basis. In turn, a customer's software administrators analyze newly published defects shortly after publication and use their expertise to assess the defect rediscovery probability and the severity of implications associated with its rediscovery. If the administrators decide the risks warrant it, they will

contact the manufacturer’s support desk requesting a special software build[26] incorporating a defect fix. This is a preventative measure against encountering this problem in the future.

Reactive requests for software fix: A customer could encounter a defect recently exposed by another customer (this is common for “regression” defects which break existing functionality) so requesting a special build from the support desk to prevent defect reencounter.

In both cases, the support desk will, after an initial assessment, relay this special build compiling and testing request to the manufacturer’s maintenance team. Large numbers of customers classifying a defect as “potentially discoverable”, may trigger an avalanche of special build requests. These requests can overload the maintenance and support personnel. We now analyze the cause of the overload and the actions needed to prevent it.

Maintenance Team: Customers may use different versions of the product on multiple platforms. Even though the source code repairing a given defect is the same, special builds will have to be tailored individually for each customer. Building and testing a special build of a large software product can take several days, consuming human and hardware resources. Therefore, the maintenance team is interested in knowing the probability of the increase in the number of requests for special builds above a certain threshold[27] in a certain timeframe as well as the total number of the requests above the threshold. We call the number of requests for special builds above a certain threshold a “*spike*”. In addition to the probability of a spike, the maintenance team is also interested in the conditional expectation of the spike’s size given its occurrence. Also of interest is the probability that the number of requests for special builds in a given timeframe will not exceed a predetermined threshold. By leveraging this data, the management of the

---

[26] We assume that the standard vehicle for delivery of fixes is through cumulative fix packs.

[27] In addition to routine requests for special builds for defects with small numbers of rediscoveries.

maintenance team can allocate personnel (based on the expected number of special builds and the average waiting time to deliver the builds) so that they can be transferred to the "special build team" on an as-needed basis, decreasing delivery time to customer.

Support Team: Once contacted by a customer with a proactive request for a special build (fixing defect of interest), a support analyst must verify if the defect can be rediscovered by the customer[28]. If the request is reactive, then the analyst has to verify that the problem is caused by this particular defect and not another one with similar symptoms. Knowing the probability and potential size of spikes in the number of requests (as well as the probability of not exceeding a certain number of calls in a given timeframe) can support management's personnel allocation, speeding diagnostics thus leading to faster transfers to the maintenance team and a decrease in the overall turnaround time. The end result is cost savings and higher customer satisfaction.

4.3.1.2 Quality Assurance Team

Maintenance and Support teams can use information about frequently rediscovered defects for tactical planning. The QA team can use this data for strategic planning to identify trends in software quality on a release-to-release basis. Frequently rediscovered defects affecting a significant percentage of the customer base relates to frequently executed common functionality. The presence of such defects suggests the QA team's inability to reproduce customer workloads in-house or its failure to execute existing test-cases covering this functionality [11]. In order to compare releases of the product, an analyst needs to find out how many defects were rediscovered at least x times for a given release[29]. The numbers of defects with high number of rediscoveries should decrease

---

[28] For example, even though the customer is using functionality affected by a given defect, the problem could be specific to a hardware platform not used by this customer.

[29] If the customer base of a software product does not change significantly from release to release, then the number of defects with a high number of rediscoveries can be directly compared. If this assumption fails then it may be beneficial to normalize the number of defects by the size of customer base and/or product usage.

from release to release. An increasing number of defects may imply a deterioration of QA processes. The QA team should analyze root causes of defects to find the actions needed to close these gaps.

4.3.1.3 Customers

Information about defect rediscovery interests customers, especially for mission-critical applications. It is known that a customer's perceived quality [12],[13],[11] is correlated with the quantity and severity of failures that the customer encounter. Therefore, comparison of the number of defects affecting a significant percentage of the customer base for various products can be used as one of the measures needed to select the "safest" product. In the next section we discuss some techniques required to answer these questions.

4.3.2 Formulation of Metrics

Based on the discussion in the previous section, stakeholders are interested in the following data:

1. The number of defects rediscovered more than certain number of times in a given timeframe;

2. The number of defects affecting a certain percentage of the customer base in a given timeframe;

3. The total number of rediscoveries for defects rediscovered more than certain number of times in a given timeframe;

4. The probability of spikes in the number of requests in a given timeframe;

5. The probability that the number of requests for special builds in a given timeframe will not exceed a certain threshold;

6. The worst-case scenario for the total number of rediscoveries;

7. The expected waiting time of customers.

To calculate these variables we build a formal probabilistic model of defect rediscoveries.

Suppose that *N* field defects are discovered independently up to time *t* with the *i*-th defect rediscovered $D_i \equiv D_i(s,t)$ times in the interval [s, t), s<t. For the sake of brevity we will use D_i and $D_i(s,t)$ interchangeably. The number of rediscoveries $R(s,t)$ between times *s* and *t* is given by

$$R(s,t) \equiv \sum_{i=1}^{N(t)} D_i(s,t). \tag{4.1}$$

Formally, a spike is defined as the situation when the total number of rediscoveries in a given timeframe [s,t] is greater than *r*:

$$R(s,t) > r. \tag{4.2}$$

The probability that the *i*-th defect will be rediscovered exactly *d* times in the interval $[s,t)$ is given by $p_i(d) \equiv P(D_i = d)$. We assume that the probability distribution of the number of rediscoveries is the same for all defects (i.e., that the D_i are identically distributed random variables).

Assuming that the number of rediscoveries lies in the range [0,∞), the probability that the number of rediscoveries of the *i*-th defect will be less than or equal to *d* is given by cumulative distribution function (cdf)

$$F_i(d) = E\left[I_{D_i \le d}\right] = P(D_i \le d) = \sum_{j=0}^{d} p_i(j), \tag{4.3}$$

where I_A is an indicator variable such that

$$I_A = \begin{cases} 1, & \text{if } A \text{ holds} \\ 0, & \text{otherwise} \end{cases}; \tag{4.4}$$

and the expected value of A is equal to probability of A:

$$E\left[I_A\right] = P(A). \tag{4.5}$$

The probability that the number of rediscoveries of the i-th defect will be greater than d is given by the decumulative distribution function:

$$\tilde{F}_i(d) = E\left[I_{D_i > d}\right] = P(D_i > d) = \sum_{j=d+1}^{\infty} p_i(j) = 1 - F_i(d). \tag{4.6}$$

The quantile function, (inverse of the cdf) $F_i^{-1}(\alpha)$ is used to determine the α quantile of a given distribution.

The expected total number of rediscoveries for the i-th defect with rediscoveries ranging between l and u is given by

$$R_i(l,u) = E\left[D_i I_{l \le D_i \le u}\right] = \sum_{j=l}^{u} j p_i(j). \tag{4.7}$$

Note that $R_i(1,\infty)$ calculates expected number of rediscoveries of the i-th defect. Armed with these instruments, we can estimate the metrics listed above.

M_1: Expected number of defects rediscovered more than certain number of times

The expected number of defects rediscovered more than d times is given by

$$M_1(d) = E\left[\sum_{i=1}^{N} I_{D_i > d}\right] = \sum_{i=1}^{N} E\left[I_{D_i > d}\right] = \sum_{i=1}^{N} \tilde{F}_i(d). \tag{4.8}$$

If all p_i are identically distributed, then (4.8) simplifies to

$$M_1(d) = \sum_{i=1}^{N} \tilde{F}_i(d) \overset{\text{i.d.}}{=} N\tilde{F}_1(d) = N\tilde{F}(d). \tag{4.9}$$

Note that we suppress indices to ease notation.

M_2: Expected number of defects affecting certain percentage of the customer base

This metric is similar to M_1. If we denote the total number of customers by C and assume that every customer rediscovers a given defect only once, then the relation between the percentage of the customer base x and number of rediscoveries d is given by

$$\tilde{d} \approx \lfloor xC/100 \rfloor, \tag{4.10}$$

where $\lfloor \bullet \rfloor$ is the floor function mapping to the next smallest integer. M_2 is calculated as

$$M_2(\tilde{d}) = \sum_{i=1}^{N} F_i(\tilde{d}) \overset{\text{i.d.}}{=} NF_1(\tilde{d}) = NF(\tilde{d}), \tag{4.11}$$

M_3: Expected total number of rediscoveries for defects with number of rediscoveries above certain threshold in a given timeframe

The expected total number of rediscoveries for a given spike is calculated as

$$\begin{aligned} M_3(d) &= E\left[\sum_{i=1}^{N} D_i I_{d \le D_i < \infty}\right] = \sum_{i=1}^{N} E\left[D_i I_{d \le D_i < \infty}\right] \\ &= \sum_{i=1}^{N} R_i(d,\infty) \overset{\text{i.d.}}{=} NR_1(d,\infty) = NR(d,\infty), \end{aligned} \tag{4.12}$$

where d is the smallest number of rediscoveries of a particular defect.

M_4: Probability of spikes in the number of requests in a given timeframe

This can be rephrased as probability that the total number of rediscoveries will exceed a certain threshold L. The calculation of this value involves two steps:

1) Find d to satisfy the equation:

$$L = E\left[\sum_{i=1}^{N} D_i I_{1\le D_i \le d}\right] = \sum_{i=1}^{N} E\left[D_i I_{1\le D_i \le d}\right] = \sum_{i=1}^{N} R_i(1,d) \overset{\text{i.d.}}{=} NR_1(1,d) = NR(1,d), \quad (4.13)$$

Since d is discrete, we will not always be able to find an integer value of d to satisfy this equality, so we look for the smallest integer d which satisfies:

$$L \le \sum_{i=1}^{N} R_i(1,d) \overset{\text{i.d.}}{=} NR_1(1,d) = NR(1,d), \quad (4.14)$$

2) After identifying d, the probability that the total number of rediscoveries will exceed L is given by

$$M_4(d) = \tilde{F}(d) = 1 - F(d). \quad (4.15)$$

M_5: Probability that the total number of rediscoveries will not exceed certain threshold

This metric is complementary to M_4 and is calculated in a similar manner. Given the number of rediscoveries d from (4.14) we calculate M_5 as

$$M_5(d) = 1 - M_4(d) = F(d). \quad (4.16)$$

M_6: Estimate of the worst case scenario for the total number of rediscoveries

This metric provides a threshold which the total number of rediscoveries will not exceed for a given probability level. The metric provides the worst case scenario of the total number of rediscoveries. For example, if the value of $M_6(0.99)$ is equal to y, then it will tell us that in 99 cases out of 100 the total number of rediscoveries will not exceed y[30].

[30] This is similar to "Value At Risk" measure used in finance

In order to obtain this value we need to identify number of rediscoveries for a given probability level α using $\lfloor F_i^{-1}(\alpha) \rfloor$. The threshold value of rediscoveries is then calculated using

$$\begin{aligned} M_6(\alpha) &= E\left[\sum_{i=1}^{N} D_i I_{1 \le D_i \le \lfloor F_i^{-1}(\alpha) \rfloor}\right] = \sum_{i=1}^{N} E\left[D_i I_{1 \le D_i \le \lfloor F_i^{-1}(\alpha) \rfloor}\right] \\ &= \sum_{i=1}^{N} R_i\left[1, \lfloor F_i^{-1}(\alpha) \rfloor\right] \overset{\text{i.d.}}{=} NR_1\left[1, \lfloor F_1^{-1}(\alpha) \rfloor\right] \\ &= NR\left[1, \lfloor F^{-1}(\alpha) \rfloor\right]. \end{aligned} \tag{4.17}$$

M_7: Expected waiting time of customers being serviced

This metric is calculated using queuing tools [14]. M_7 depends on the distributions governing service time, requests' inter-arrival time and number of personnel allocated to handle these requests. The metric's formula will depend on the form of distributions governing the queue.

Let us look at the application of the metrics. Metrics M_1 and M_2 can be used by QA and customers to calculate the number of defects injected in common functionality (M_1) and identify defects affecting a certain fraction of the customer base (M_2) as discussed in Sections 4.3.1.2 and 4.3.1.3.

Metric M_3 helps to estimate the total number of rediscoveries for frequently discovered defects and the potential contribution of the frequently rediscovered defects to the overall load of support and maintenance teams.

Metrics $M_{4\text{-}7}$ can be used to address issues described in Section 4.3.1.1 and help in resource allocation of the service and maintenance teams.

Metrics $M_{4\text{-}6}$ may also be used for resource allocation as follows: A manager responsible for resource allocation knows the amount of available personnel, denoted by A, and,

based on historical data, the average amount of service (special build) requests that support (maintenance) person can process per unit time, denoted by μ.

A simple estimate[31] of the overall amount of service (special build) requests, denoted by Q, that can be processed by personnel in a given timeframe T is given by

$$Q = A\mu T. \quad (4.18)$$

The manager can then use $M_4(Q)$ or $M_5(Q)$ to get an estimate of the probability that the support of maintenance team will be able to handle the volume of requests Q.

The manager can examine the resource allocation task from the opposite perspective: instead of calculating the probability of handling requests by employees, she can calculate the number of service or special build requests that will not exceed $M_6(\alpha)$ at confidence level α. We can obtain the amount of personnel A needed to handle this workload by inverting (4.18):

$$A = M_6(\alpha)/(\mu T). \quad (4.19)$$

Note that stationary processes should be used if metrics $M_{4\text{-}7}$ are used for forecast-related management decisions. In order to calculate the metrics $M_{1\text{-}6}$ we also need an estimate of the total number of defects. There exists a variety of methods that can be used to estimate this value, see [15] for review of the methods. Detailed discussion of these techniques is beyond the scope of this paper.

4.4 Case Study

In this case study we use defect discovery data for a set of components of four consecutive releases of a large scale enterprise software. To preserve data confidentiality,

---

[31] This estimate does not account for request inter-arrival times. A better estimate can be obtained using metric M_7

the dataset is scaled and rounded. Also, we assume that the customer base size remains constant across all four releases.

Figure 18 depicts *N*(t), the cumulative number of defects encountered up to *t* years after the product has shipped. The total number of rediscoveries from time 0 (general availability (GA) date of the product to be shipped to the field) to time *t*, *R*(0,*t*), is shown in Figure 19. The age of the releases in the field varies from 5 years for v.1 to 2 years for v.4 because v.4 was released about three years after v.1

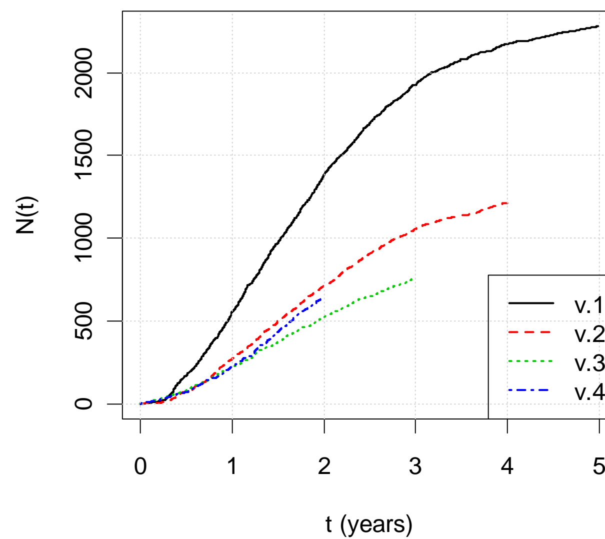

Figure 18. *N*(*t*): total number of defects discovered up to time *t*.

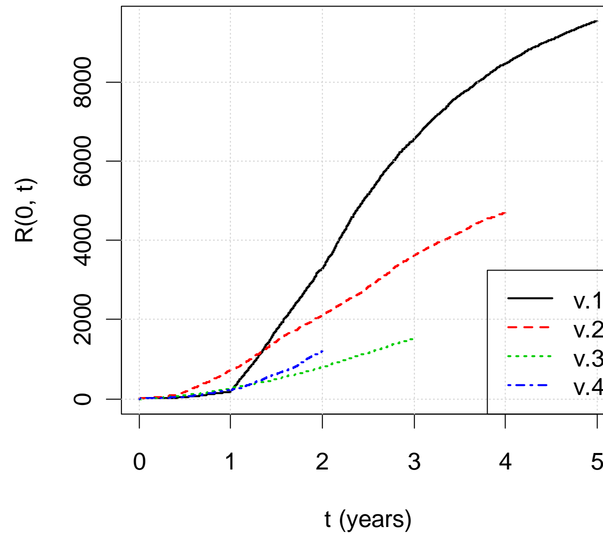

Figure 19. ***R*(0,*t*): total number of rediscoveries up to time *t*.**

Metrics $M_{1\text{-}6}$ rely on the distribution of the number of rediscoveries per defect D_i (Section 4.3.2). In the same section, to simplify formulas for $M_{1\text{-}6}$, we assumed that D_i are identically distributed so must specify the distribution of D_i. Without loss of generality, we split the D_i data for every release into yearly time intervals $D_i(t, t+1)$, where $t = 0 \ldots 4$ (if the data is present for a given release).

This split is reasonable in practice. Resource planning (metrics $M_{3\text{-}6}$) is performed for a fairly short future time interval; one year or less being common planning horizons. Metrics $M_{1\text{-}2}$ focus on measuring general quality of the product and would benefit from the information about rediscoveries over the complete lifecycle of a product in the field. However, it is also critical to identify issues with QA processes early, so that actions can be taken to improve QA processes of releases under development. Since the lifespan of an enterprise software product can often reach a decade or more, it would not be practical to wait such a long time to obtain information.

4.4.1 Finding a Suitable Distribution

In order to find an analytic distribution that would be able to fit each of the yearly datasets, we use an L-moments ratio diagram [16]. This diagram is a goodness-of-fit tool to determine the probability distribution of the data. The L-moments are chosen since they are less biased and are less sensitive to outliers than ordinary moments [17],[16].

The diagram is shown in Figure 20 and the hollow circles denote each of the yearly datasets of D_i. The diagram shows the fits of the following widely used distributions [18]: Exponential (EXP), Normal (NOR), Gamma (GUM), Rayleigh (RAY), Uniform (UNI), Generalized Extreme Value (GEV), Generalized Logistic (GLO), Generalized Normal (GNO), Generalized Pareto (GPA), generalization of the Power Law, Pearson Type III (PE3), and Kappa (KAP). The diagram shows that the data is best approximated by a Kappa distribution as all data lie in the Kappa applicability space[32], with the Pearson Type III distribution the second best choice (data points lie around PE3 L-moments ratio line).

The analysis procedure is adequately shown even if we limit the scope of the analysis to the four datasets of $D_i(1,2)$ showing rediscovery data for the second year of each release. We note that due to heavy tails, the exponential distribution does not provide an adequate fit to the data. Based on the data from the L-moments ratio diagram, we fit the data using the two best performers: Pearson Type III and Kappa distributions. The QQ-plots showing goodness of fit are shown, accordingly, in Figure 21 and Figure 22. Based on Akaike's information criterion (AIC) [19] the Kappa distribution provides a better fit[33] than Pearson Type III for three datasets out of four (see Table 11).

---

[32] The Kappa distribution's applicability space is a plane bounded by GLO and "Theoretical limit" L-moments ratio lines [16] on Figure 20.

[33] The lower the value of AIC – the better the fit.

Table 11. AIC

Distribution	v.1	v.2	v.3	v.4
PE3	12397	4481	2069	4797
KAP	11277	4309	2390	4304
Compound KAP	9392	4283	2934	4238

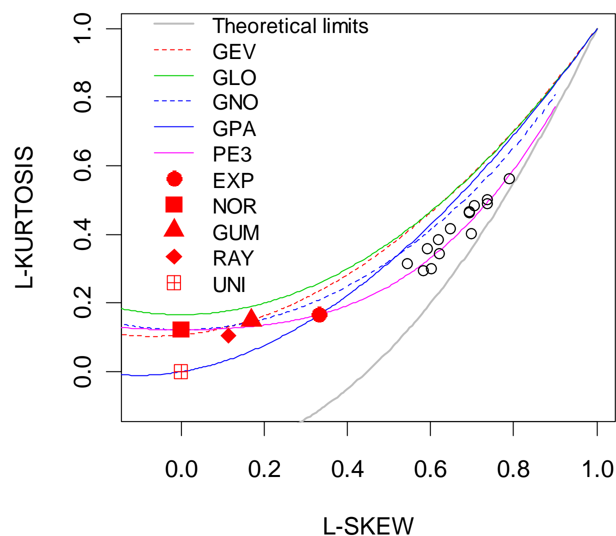

Figure 20. L-moments ratio diagram of D_i for all releases per year (years 1 – 5). The hollow circles denote each of the yearly datasets of D_i. The diagram shows the fits of the following distributions: Exponential (EXP), Normal (NOR), Gamma (GUM), Rayleigh (RAY), Uniform (UNI), Generalized Extreme Value (GEV), Generalized Logistic (GLO), Generalized Normal (GNO), Generalized Pareto (GPA), generalization of the Power Law, Pearson Type III (PE3), and Kappa (KAP). Kappa distribution applicability space is a plane bounded by GLO distribution line above and the "Theoretical limits" line below and is not shown on the legend. Based on this figure, Kappa distirbution is the only one that is applicable to modeling each of the datasets.

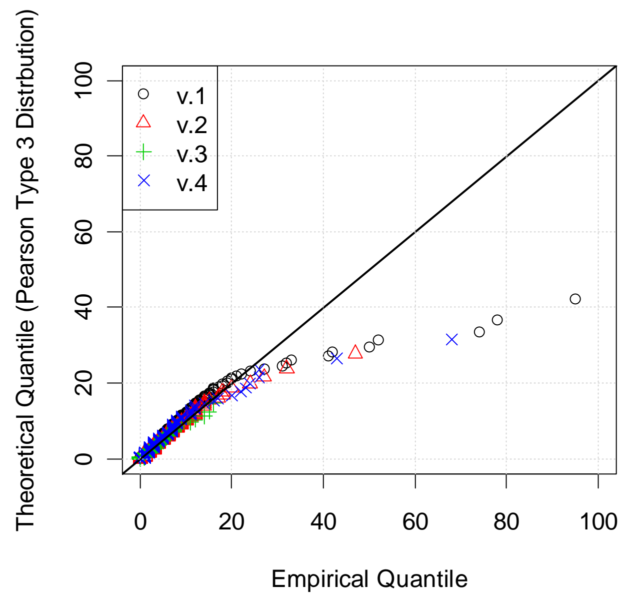

Figure 21. QQ plot of the empirical vs. PE3 distributions' quantiles.

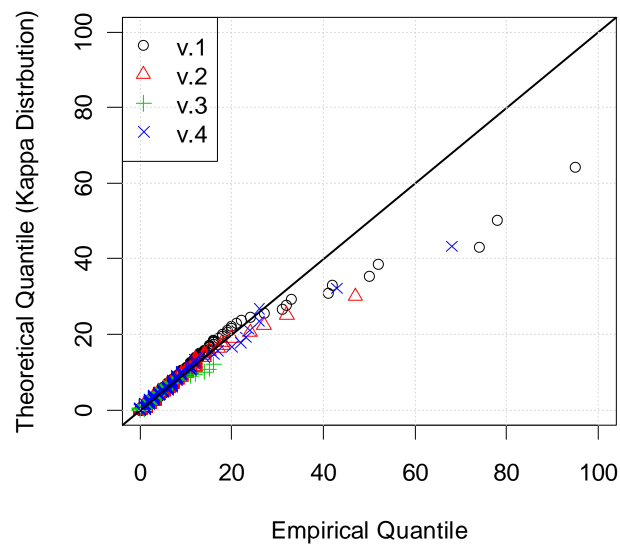

Figure 22. QQ plot of the empirical vs. KAP distributions' quantiles.

However, Figure 22 suggests that even the Kappa distribution isn't sufficiently flexible to fit both left and right tails of the empirical distribution. In order to overcome this obstacle we resort to a compound Kappa distribution.

The Kappa distribution [16] is a flexible 4-parameter distribution suited for fitting heavy-tail data. This distribution contains the Exponential, Weibull, Generalized Extreme Value, and Generalize Pareto distributions as special cases. Its cdf is:

$$F(x) = \left\{ 1 - h\left[1 - \frac{\kappa(x-\xi)}{\alpha} \right]^{1/\kappa} \right\}^{1/h}. \qquad (4.20)$$

The parameters ξ , α, and κ and h describe location, scale, and shape, respectively. The associate quantile function is:

$$F^{-1}(u) = \xi + \frac{\alpha}{\kappa}\left[1 - \left(\frac{1-u^h}{h} \right)^{\kappa} \right]. \qquad (4.21)$$

The first Kappa distribution with cdf F_a, fits the left tail of the dataset (in the range [0, ρ]) and the second with cdf F_b fits the right tail in the range (ρ, ∞). We select these partition points, ρ, for each of the four datasets by minimizing the sum of squares of the residuals between fitted and empirical data. For other techniques see [19]. Table 12 presents values of ρ. F_a and F_b are fitted independently; the resulting cumulative distribution function looks like:

$$F_c(d) = \begin{cases} ww_1 F_a(d), D \le \rho \\ w\left[w_1 F_a(\rho) + w_2 F_b(d) \right], D > \rho \end{cases}, \qquad (4.22)$$

where w, w_1, w_2 are the normalization constants

$$
\begin{aligned}
&w_1 \equiv \text{ ecdf}(\rho), \\
&w_2 = 1 - w_1, \\
&w = \left[w_1 F_a(\rho) + w_2 F_b(\infty) \right]^{-1} = \left[w_1 F_a(\rho) + w_2 \right]^{-1},
\end{aligned}
\tag{4.23}
$$

where ecdf is the empirical distribution function. We use the Weibull form [20] of the empirical distribution function: given a vector of observations y sorted in ascending order, with sample size n, the unbiased non-exceedance probability of the i-th observation is given by:

$$
\text{ecdf}\,(i) = i / (n+1). \tag{4.24}
$$

The quantile function of the compound distribution can be obtained by inverting (4.22):

$$
F_c^{-1}(u) = \xi_{\omega} + \frac{\alpha_{\omega}}{\kappa_{\omega}} \left[1 - \left(\frac{1 - z(u)^{h_w}}{h_{\omega}} \right)^{\kappa_{\omega}} \right], \tag{4.25}
$$

where

$$
z(u) = \begin{cases} u / (w w_1), \ u \le w w_1 F_a(\rho) \\ \left[u - w w_1 F_a(\rho) \right] / w w_2, \ u > w w_1 F_a(\rho) \end{cases}.
$$

The parameters' lower index, ω, specifies their affiliation with the first (a) or second (b) Kappa distributions; $\omega = a$ if $u \le w w_1 F_a(\rho)$ and $\omega = b$ otherwise.

Table 12. Values of variables

Variable	v.1	v.2	v.3	v.4
ρ	15	15	8	10
$M_1(10)$	65.52	26.27	6.02	20.64
R(1,2)/N(2)	2.24	1.76	1.02	1.53

The goodness of fit of the compound distribution is shown on QQ-plot in Figure 23. The QQ-plot suggests that the compound distribution provides a good fit to the underlying data. In addition, based on the AIC data given in Table 11, the compound Kappa distribution provides better fit than the Kappa distribution for three datasets out of four.

Unfortunately, we cannot use the Kolmogorov-Smirnov or chi-squared tests due to the large number of tied observations, as Figure 24 shows.

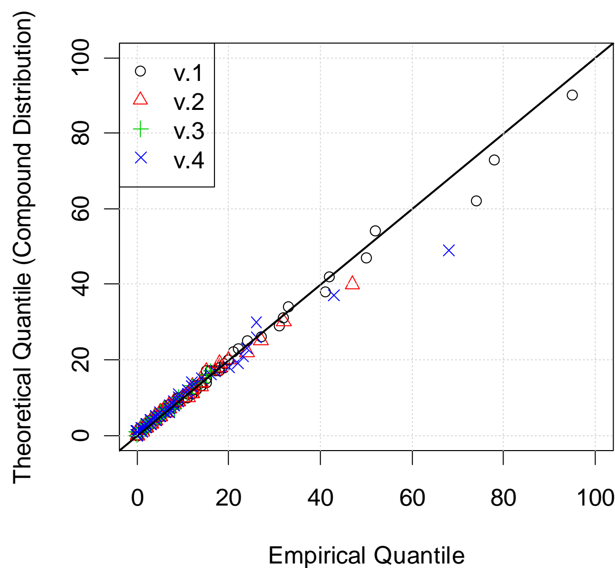

Figure 23. QQ plot of the empirical vs. Compound distributions' quantiles.

All of the distributions above are fitted to the data using the method of L-moments [16]. We have chosen this technique over the classical method of moments due to a more accurate estimate of the distribution's right tail [16],[17]. Having established the machinery for estimation of the metrics, we proceed to examples of the analysis and usage of the metrics themselves.

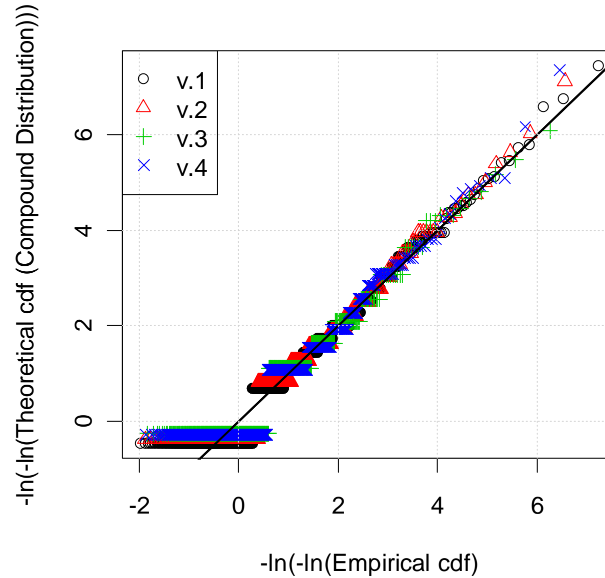

Figure 24. Plot of the empirical cdf vs. Compound Kappa distribution theoretic cdf.

4.4.2 Application of the Metrics

The application section is divided into two parts. Section 4.4.2.1 focuses on software quality metrics, while Section 4.4.2.2 concentrates on resource allocation-related metrics.

4.4.2.1 Analysis of Software Quality

As discussed in Sections 4.3.1. and 4.3.2, metrics M_1, defined by equation (4.9), and M_2, equation (4.11), can be used to identify potential issues with QA processes and to help customers find the "safest" product. Figure 25 plots M_1 against the number of rediscoveries *d.* The plot shows that from v.1 to v.3 the number of defects rediscovered more than d times decreased for all values of d. However, the value of the metric went up for v.4:

$$M_1^{v.1} < M_1^{v.2} < M_1^{v.3} < M_1^{v.4}.$$

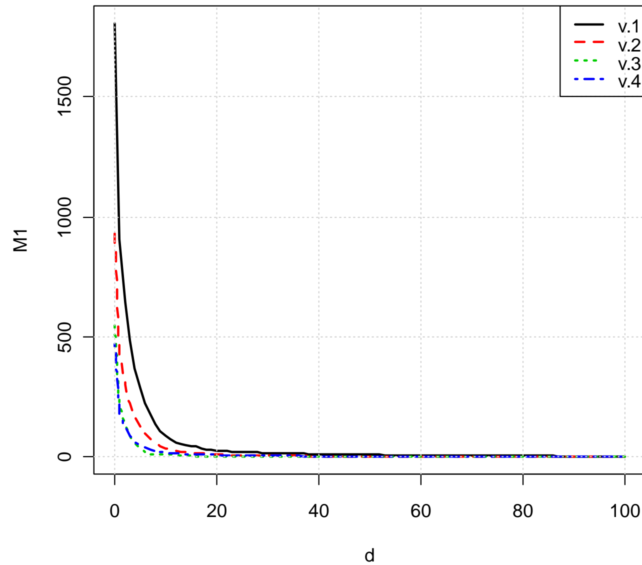

Figure 25. M_1: expected number of defects rediscovered more than *d* times during the 2nd year after GA date.

This ordering becomes especially obvious if we look at values of M_1 for a specific number of rediscoveries (for example, d=10[34]) for all releases (shown in Table 12).

This information suggests that the quality $\mathbb{Q}$ of QA processes went down in the last release:

$$\mathbb{Q}(v.1) < \mathbb{Q}(v.2) < \mathbb{Q}(v.4) < \mathbb{Q}(v.3). \qquad (4.26)$$

Before making this conclusion we should look at other quality attributes of the software. The number of rediscoveries per defect: $R(1,2) / N(2)$, given in Table 12; the ordering of

[34] We pick this number arbitrarily; an analyst can pick this threshold value based on their expertise on problematic levels of rediscoveries in their organization.

the total number of defects (Figure 18), and their rediscoveries (Figure 19) in the second year concur our hypothesis (4.26).

Based on this conclusion, an analyst needs to identify gaps in QA processes by analyzing reasons for the defects' injection and the defects' escape to the field. Upon identifying the gaps, actions should be derived and taken to prevent injection and escape of defects in future releases of the software. Additional data can be extracted by focusing the analysis on subsets of data grouped by testing team, functionality, etc.

Since we assume that the number of customers remains constant for all four releases Eq. (4.10) implies metric M_2 is a scaled version of M_1. Therefore, the number of defects affecting a certain fraction of the customer base is larger for v.4 than for v.3. At this stage a customer should perform risk-benefit analysis: would the value of v.4's new features outweigh the increased risk of encountering defects[35]. The customer can perform additional analysis by looking at M_2 for a specific subset of defects that may critically affect operations, e.g., defects in critical functionality leading to a software crash, while omitting defects that are related to functionality not used by this particular customer.

4.4.2.2 Resource Allocation

Application of metrics $M_{3\text{-}7}$ for resource allocation is discussed in Sections 4.3.1.1 and 4.3.2.

A few examples of the value of these metrics are as follows. Suppose that the maintenance team manager needs to analyze recourse allocation for building rediscovery-related special builds for v.4 during the third year of service. Currently, the manager has 8 people allocated to this task so k=8. Given an available fix, the manager knows that the average time for a team member to create, test, and ship a special build is two days: a

[35] The complete analysis should include additional factors, such as software cost and support lifespan

person can handle on average $\mu = 250/2 = 125$ requests per year[36]. Based on historical data, we know that the process governing the arrival of rediscoveries during the third year is the same as during the second year. Therefore, we can use the data from the second year to get resource allocation estimates for the third year.

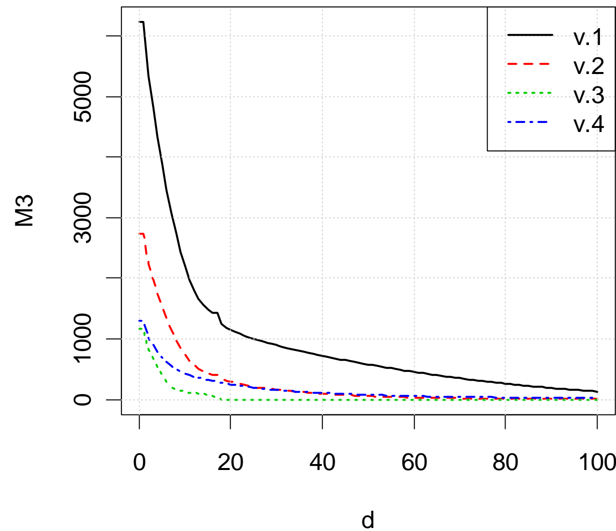

Figure 26. M_3: expected total number of rediscoveries for defects with number of rediscoveries above *d* during the 2nd year after GA date.

To put the importance of this team into perspective, the manager needs to know the fraction of rediscovery-related special builds compared to the total number of requests for special builds. The total expected number of rediscoveries is given by $M_3(1)$ (Eq. (4.12)) and shown in Figure 26. For simplicity, we use the number of defects discovered during the second year as an estimate of the number of discoveries during the third year. In this case (based on Figure 18) the expected number of discovered defects during the third

[36] Assuming 250 working days per year.

year is approximately equal to $N(2) - N(1) = 417$. $M_3(1)$ for v.4 is equal to 1299. The fraction of the total number of request for special builds related to rediscovered defects is 1299 / (1299 + 417) ≈ 0.76. This team will handle a significant portion of the overall number of requests and, therefore, allocation of the personnel for this team can be critical.

Equation (4.18) can be used to get the average number of requests that the team can handle per year:

$$Q = k\mu \mathrm{T} = 8 \times 125 \times 1 = 1000. \qquad (4.27)$$

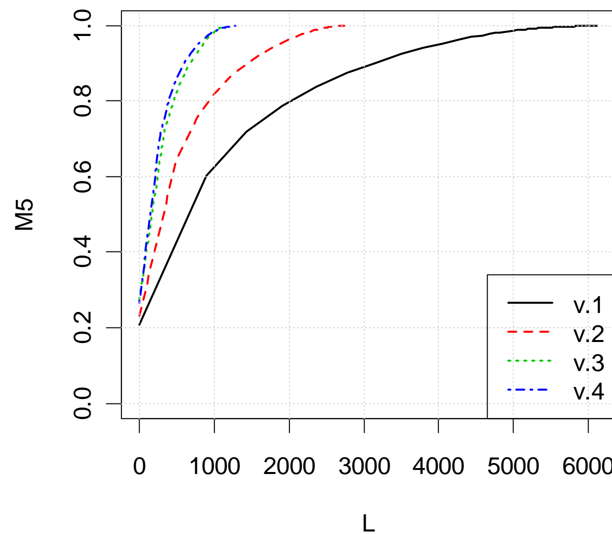

Figure 27. M_5: probability that the total number of rediscoveries will not exceed *L* during the 2nd year after GA date.

Metric $M_5(Q)$ (Eq. (4.16)) is the probability that the number of requests will not exceed Q. Based on Figure 27, $M_5(Q) = M_5(1000) \approx 0.984$. This value can be interpreted as follows: in the hypothetical case of the software being in service for 1000 years (and the arrival of rediscoveries being stationary) in 984 years out of 1000 a team of 8 people

would be able to handle the requests, in 16 years out of 1000 the number of requests would be larger than this team can handle.

What if the manager would like to know how many people is needed to handle requests in 999 years out of 1000? By using Equation (4.19) and Figure 28, the number of people needed to handle these requests is equal to:

$$k = \frac{M_6(\alpha)}{\mu T} = \frac{M_6(0.999)}{125 \times 1} \approx \frac{1245}{125} \approx 10. \qquad (4.28)$$

This suggests that the manager should allocate two additional team members to handle this extreme case.

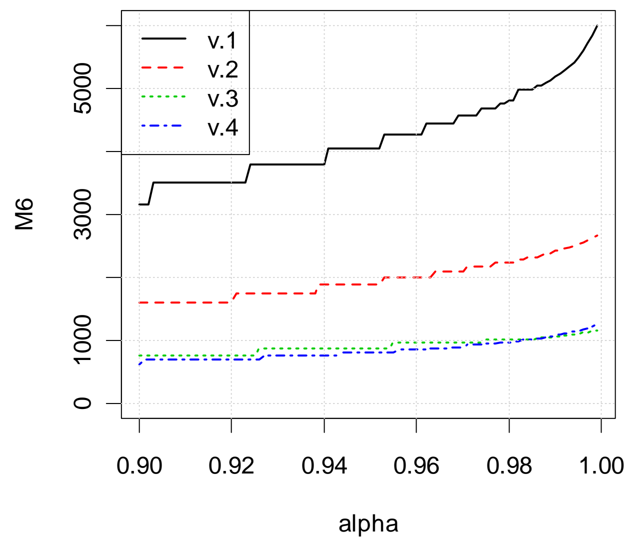

Figure 28. Estimate that the total number of rediscoveries will not exceed M_6 with confidence level α.

So far we did not consider the amount of time customers must wait to get their special build. If, at a certain time, the maintenance team receives an avalanche of requests, the

customers will have to wait for a long time to obtain their special builds. In order to obtain the expected waiting time, W, we need to model this queue [15] to consider 1) the distribution of request inter-arrival times. [14]; and 2) the time to complete service. We assume that the process is stationary, the queue is "First in, First out", and the service times are exponentially distributed with mean service time equal to $1/\mu = 1/125 = 0.008$ years. The empirical average number of requests for special builds of v.4 during the second year is λ = 982 requests per year. The distribution of inter-arrival times for v.4 (second year) is given in Figure 29. We could not find an analytic distribution providing good fit to the data. Due to this fact, we pick a queuing model denoted, using Kendall's notation [14], as G/M/k:

- G: **g**eneral distribution of inter-arrival requests. In our case we will use the empirical distribution in Figure 29,
- M: exponential distribution of service times,
- k: number of team members handling the requests.

Details of the model are given in [14].

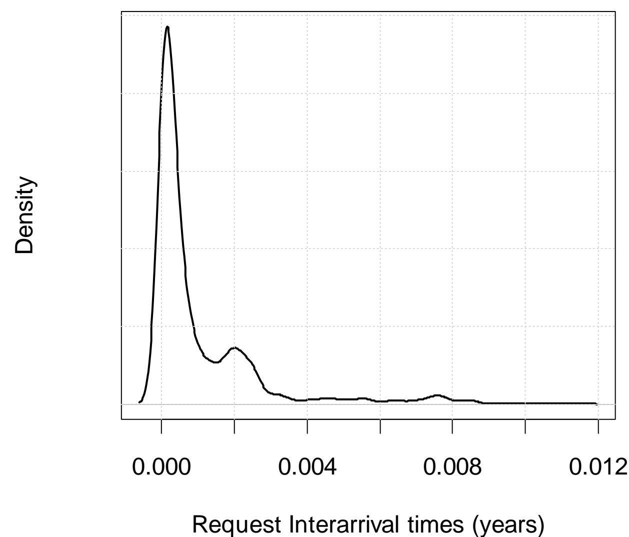

Figure 29. Density of requests inter-arrival times for v.4, second year.

Complementary to W, we calculate the percentage of the overall working time the team members spend generating special builds:

$$b = \lambda / (k\mu) \times 100. \tag{4.29}$$

For example, average busy time for 8 team members is

$$b = 982 / (8 \times 125) \times 100 \approx 98.2\%.$$

Model results are given in Table 13 showing that 8 team members can handle service requests. However, average waiting time will be 26.3 working days, which may be unacceptably long. Increasing the team to 10 decreases W to 2.9 days and a 12 member team further reduces W to 2.2 days. However the associated busy time of team members drops from 98.2% for 8 team members to 65.5% for 12 team members.

Table 13. Results of the G/M/k model for v.4, second year.

Number of team members k	Average waiting time W		Percent of the time the team members are busy (b)
	in years	in working days	
8	0.1052	26.3	98.2%
9	0.0170	4.3	87.3%
10	0.0115	2.9	78.6%
11	0.0097	2.4	71.4%
12	0.0089	2.2	65.5%

With this information, the manager can now select the optimal team size and plan additional tasks for the team members to fill their free time. The analysis of support personnel allocation is performed in a similar manner.

4.4.3 Threats to Validity

The underreporting of problems by customers can skew the dataset making the right tail of the D_i distribution heavier. Two main types of defects are not reported to the service desk: 1) defects with low severity with obvious workarounds and 2) non-reproducible defects that the customers encounter during coincidences of multiple events which disappear after restarting the software.

Underreporting may bias the analysis of actual software quality (Section 4.4.2.1). However, bias will be consistent across releases as long as underreporting is. Underreporting will not affect resource allocation processes (Section 4.4.2.2), since service and maintenance teams are interested in prediction of the actual number of support or special build requests. For them, a bug that is not reported does not exist.

4.5 Conclusions

Defect rediscovery is an important problem affecting both software manufacturers and customers. We have introduced a set of practical metrics designed to assess risks associated with defect rediscovery. The metrics can help the QA team with performance analysis of QA processes. They aid support and maintenance teams with resource allocation and with estimation of risk associated with under-staffing. Finally, the metrics provide customers with information on quality of various software products to help identify products best suited for their needs. The metrics can be applied to any defect rediscovery dataset and are distribution-independent. We believe that these metrics are applicable to other software products. We also presented a validation case study showing application of the metrics to industrial data.

References

[1] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995, pp. 424-433.

[2] A. Mockus and D. Weiss, "Interval quality: relating customer-perceived quality to process quality," *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 723-732.

[3] C.T. Baker, "Effects of Field Service on Software Reliability," *IEEE Trans. Softw. Eng.*, vol. 14, 1988, pp. 254-258.

[4] S.S. Gokhale and R.E. Mullen, "Queuing Models for Field Defect Resolution Process," *Proceedings of the 17th International Symposium on Software Reliability Engineering*, 2006, pp. 353-362.

[5] S.S. Gokhale and R. Mullen, "Software defect repair times: a multiplicative model," *Proceedings of the 4th international workshop on Predictor models in software engineering*, 2008, pp. 93-100.

[6] R. Hewett and P. Kijsanayothin, "On modeling software defect repair time," *Empirical Softw. Eng.*, vol. 14, 2009, pp. 165-186.

[7] S. Wagner and H. Fischer, "A Software Reliability Model Based on a Geometric Sequence of Failure Rates," *Reliable Software Technologies – Ada-Europe 2006*, 2006, pp. 143-154.

[8] E.N. Adams, "Optimizing preventive service of software products," *IBM J. Res. Dev.*, vol. 28, 1984, pp. 2-14.

[9] R.E. Mullen and S.S. Gokhale, "Software Defect Rediscoveries: A Discrete Lognormal Model," *Software Reliability Engineering, International Symposium on*, 2005, pp. 203-212.

[10] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, 2008, pp. 1-26.

[11] A.P. Wood, "Software Reliability from the Customer View," *Computer*, vol. 36, 2003, pp. 37-42.

[12] A. Mockus, P. Zhang, and P.L. Li, "Predictors of customer perceived software quality," *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 225-233.

[13] M. Buckley and R. Chillarege, "Discovering relationships between service and customer satisfaction," *Proceedings of the International Conference on Software Maintenance*, 1995, p. 192.

[14] D. Gross, J.F. Shortle, J.M. Thompson, and C.M. Harris, *Fundamentals of Queueing Theory*, Wiley-Interscience, 2008.

[15] P.L. Li, M. Shaw, and J. Herbsleb, "Selecting a defect prediction model for maintenance resource planning and software insurance," *Proceedings of the Fifth Workshop on Economics-Driven Software Research*, 2003, pp. 32-37.

[16] J.R.M. Hosking, "The four-parameter kappa distribution," *IBM J. Res. Dev.*, vol. 38, 1994, pp. 251-258.

[17] R.M. Vogel and N.M. Fennessey, "L Moment Diagrams Should Replace Product Moment Diagrams," vol. 29, 1993, pp. 1745-1752.

[18] N.L. Johnson, S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions, Vol. 1*, Wiley-Interscience, 1994.

[19] H. Akaike, "A new look at the statistical model identification," *IEEE Trans. on Automatic Control,* , vol. 19, 1974, pp. 716-723.

[20] L. Makkonen, "Bringing Closure to the Plotting Position Controversy," *Communications in Statistics - Theory and Methods*, vol. 37, 2008, pp. 460-467.

Chapter 5

5 Selection of Customers for Operational and Usage Profiling

Operational and usage profiles collected from customers provide developers and testers with valuable quantitative information on usage patterns of software being developed. Unfortunately, gathering such profiles from a large set of customers can be challenging due to time and resource constraints. In this chapter we propose to use information about defects that customers found to narrow down a list of candidate customers to profile. We present a technique for selection and prioritization of a minimal set of customers for operational and usage profiling to cover a certain set of defects. The technique optimally selects a minimal set of customers for profiling and, once the set is identified, prioritizes the customers within the minimal set. We describe a validation case study confirming that this approach is scalable for a large customer base. Analysis results can then be used to close gaps in testing coverage and to improve the maintenance process.

5.1 Introduction

The number of execution paths grows combinatorially with software size [6]. Therefore, it is almost impossible to cover all execution paths because of development complexity, time and resource constraints. Analysis of field defects allows us to identify major testing gaps and close them, thereby improving the testing quality of future releases. If a customer reports a lot of defects, we can deduce that customers' usage patterns cover a lot of execution paths that had not been covered in-house. We can profile these customers (gather data, workloads, etc.) and incorporate this information in testing scenarios.

If software is used by a large number of customers, we cannot possibly profile all of them. Moreover, some of the customers might not provide us access to their system or

data for confidentiality, statutory, or contractual reasons[37]. Fortunately, information about defects that each customer discovers provides us with a list of problematic areas covered by each particular customer. Analysis of this data can help us to narrow down a list of "interesting" customers that we need to profile.

This chapter proposes a technique that 1) optimally selects a minimal set of customers for profiling and 2) once the set is identified, prioritizes the customers within the minimal set. Results from a case study we conducted (described in this paper) show that this approach is scalable for use in large-scale software development environments.

This approach helps to

- reduce defect escapes to the field[38],
- close gaps in test coverage, and
- prioritize test coverage.

Let us look at each of the benefits in detail.

Reduce defect escapes to the field: The goal is to reduce and or eliminate software defects that escape to the field. This has a direct impact on improving the quality and reducing the maintenance cost of software in the field. Improving either or both of these will increase customer satisfaction and reduce maintenance and development cost over the lifetime of the product. This work discusses a method to identify defects that need to be addressed in order to close the gaps in coverage, both in the test and development process in order to reduce defect escapes to the field. We also provide a method to

---

[37] For example, a database may contain sensitive information such as confidential financial or research data.

[38] Field defects are defects found by customers during post-release phase.

identify customers for operational profiling[39] [7] to assist in gap analysis. Within these two methods we will discuss how to prioritize and analyze the field defects.

Close gaps in test coverage: Field defects are a result of missed coverage from all development processes. These can include escapes from design, code, test and documentation reviews. The test escapes can occur in Unit, Function, System, Integration, Regression, Alpha and Beta testing. The goal is to close these coverage gaps by adding test coverage and/or process changes. The pervasiveness of a field defect will be used to identify defects that are hit by many customers and also to identify a set of customers that have a high number of defects that are pervasive or have been hit by many other customers. The latter will be used to identify customers who would be candidates for operational profiling.

Operational profile test selection puts more emphasis on likely execution paths [7]. This may lead to exclusion of critical but infrequently executed paths (e.g., disaster recovery functionality). Coverage gaps identified during analysis of field defects provide us with objective picture of gaps' "importance". The larger the number of discoveries of a particular defect, the more important is the execution path associated with the defect.

Prioritize test coverage: One of the problems in software testing is determining what has to be tested and the prioritization of this testing through various test processes [2]. Once usage and profiling information is gathered from customers, these data should be incorporated into in-house test cases and scenarios to improve overall code coverage. Workloads that are associated with a large number of frequently discovered defects should be given higher priority than those that are associated with a small number of infrequently discovered defects.

---

[39] An operational profile is a set of operations that a software system performs along with associated probabilities of use.

The chapter is structured as follows: Section 5.2 reviews relevant work; Section 5.3 explains dimensions used for qualitative customer profiling; Section 5.4 details quantitative customer selection technique; Section 5.5 describes a case study validating our approach. Finally, Section 5.6 provides conclusions and future work.

5.2 Related Work

Operational profile development involves the following steps [7]:

1. Customer profile,
2. User profile,
3. System-mode profile,
4. Functional profile,
5. Operational profile,
6. Test selection.

Researchers focus on various aspects of operational profiling development: e.g., data gathering [4], extension of data captured during profiling [3], test selection [10], and reliability estimation [9]. However, to the best of our knowledge, no work has been done in the area of selection of customers for profiling. We describe our approach for prioritization of customers in the next section.

5.3 Qualitative Analysis Of Customers

Let us analyze customers qualitatively. We prioritize customers for profiling using two dimensions:

1. Total number of defects found by a given customer,
2. Average number of discoveries per defect found by a given customer.

Prioritization criteria are described in Table 14.

We will go over the criteria for each of the four permutations (quadrants) in detail

- **LL**: The customer finds a small number of defects that are rarely discovered by others. Incorporation of this customer's usage allows us to close a small number of gaps in testing of infrequently executed paths.
- **HL**: The customer finds a large number of defects that are rarely discovered by others. This customer can be considered "unique". Incorporation of a customer's usage allows us to close a large number of gaps in testing of infrequently executed paths.
- **LH**: The customer finds a small number of defects that are often discovered by others. These defects are clearly development, test process, and test coverage misses. The defects need to be addressed as they are interesting from a test process perspective. However, incorporation of this customer's usage allows us to close only a small number of gaps in testing of commonly executed paths.
- **HH**: The customer finds a lot of defects that are also found by other customers. Incorporation of this customer's usage allows us to close a large number of gaps in testing of commonly executed paths.

Note that we cluster the data in the four quadrants without specifying explicit thresholds for "low" and "high" values. The actual values of the thresholds will depend on the underlying data, such as total number of customers and defects. This approach gives a high level "taste" of the product's quality.

Suppose we have identified two customers who discovered a large number of defects that are also frequently discovered by other customers. We now need to make sure that the lists of defects discovered by these two customers do not overlap significantly: otherwise, we will be duplicating our effort.

Table 14. Customer prioritization criteria

Total number of defects found by a given customer	Average number of discoveries per defect found by a given customer	
	Low	High
Low	LL: Not interesting from profiling perspective	LH: Not interesting from profiling perspective
High	HL: Potential candidate for profiling	HH: Ideal candidate for profiling

Manual selection of customers using a prioritization schema described above can become cumbersome if a product is used by thousands of customers discovering hundreds of defects. Therefore, conversion of this qualitative technique to quantitative domain is difficult. We need to find a quantitative technique that will allow us to minimize the number of customers for profiling while maximizing the total number of discovered defects. Once a minimal set of customers are identified, we can prioritize them by the total number of covered defects. Details of this approach are given in the next section.

5.4 CUSTOMER SELECTION TECHNIQUE

Manual selection of customers using the prioritization schema described above can become cumbersome if a product is used by thousands of customers discovering hundreds of defects. We need to find quantitative techniques that will allow us to:

1. Minimize the number of customers for profiling while maximizing the total number of discovered defects;
2. Prioritize a minimal set of customers (once identified) by the total number of discovered defects per customer.

We describe a minimization technique in Section 5.4.1 and a prioritization technique in Section 5.4.2.

5.4.1 Minimization of Customer Set

In order to minimize a set of customers we propose to formulate this task as a Binary Integer Programming (BIP) problem [8]. We want to identify a minimal set of customers that discovered all defects.

Formally, we need to

$$
\begin{aligned}
&\text{Minimize } w_1c_1 + w_2c_2 + \ldots + w_Nc_N,\\
&\text{subject to}\\
&d_1 : p_{1,1}c_1 + p_{1,2}c_2 + \ldots + p_{1,N}c_N \geq 1,\\
&d_2 : p_{2,1}c_1 + p_{2,2}c_2 + \ldots + p_{2,N}c_N \geq 1,\\
&\ldots\\
&d_M : p_{M,1}c_1 + p_{M,2}c_2 + \ldots + p_{M,N}c_N \geq 1,\\
&\text{binary variables: } c_1, c_2, \ldots, c_N;
\end{aligned}
\tag{5.1}
$$

where

- $N \equiv$ total number of customers;
- $M \equiv$ total number of defects;
- $c_i \equiv$ i-th customer ($i = 1 \ldots N$), $c_i = 1$ if the i-th customer is included in the minimal set of customers to profile and is 0 otherwise;
- $w_i \equiv$ i-th customer weight;
- $d_j \equiv$ j-th defect (j=1…M);
- $p_{i,j} \equiv$ binary variable showing discovery of the j-th defect by the i-th customer, $p_{i,j}$=1 if the i-th customer discovered the j-th defect and is 0 otherwise.

If we want to emphasize "importance" of the i-th customer, then we should increase weight w_i relative to the weight of the remaining customers. For example, w_i can be proportional to the difficulty of gathering information from the i-th customer and inversely proportional to the average number of discoveries per defect found by the i-th customer. If all customers are considered equal then w_i=1 for all i.

In short form (5.1) can be written as

$$
\begin{aligned}
&\text{Minimize } w^T c,\\
&\text{subject to } pc \geq 1,\\
&\text{binary variables: } c.
\end{aligned}
\tag{5.2}
$$

This approach should provide us with the optimal solution [8]. In general, solution of BIP problems is NP-hard. However, if a constraint matrix p is totally unimodular[40] and the right hand side of constraints consists of integer values, then the problems can be solved efficiently [8]. Our problem formulation falls into this category of BIP problems.

5.4.1.1 Example of Selection of the Minimal Set of Customers for Profiling

Suppose we have four customers (c_1, c_2, c_3, and c_4). The customers discovered five defects (d_1, d_2, d_3, d_4, and d_5) in total. Their discoveries are summarized in Table 15.

We assume that all the customers are of equal importance and w=1. Equation (5.1) becomes

$$\begin{aligned}
&\text{Minimize } c_1 + c_2 + c_3 + c_4, \\
&\text{subject to} \\
&d_1 : c_2 \geq 1, \\
&d_2 : c_1 + c_2 + c_3 \geq 1, \\
&d_3 : c_1 + c_3 \geq 1, \\
&d_4 : c_3 + c_4 \geq 1, \\
&d_5 : c_4 \geq 1, \\
&\text{binary variables: } c_1, c_2, c_3, c_4.
\end{aligned} \tag{5.3}$$

The solution to this problem is c_1=0, c_2=1, c_3=1, and c_4=1; i.e., the minimal set of customers for profiling that cover all defects is $\{c_2, c_3, c_4\}$.

Once a minimal set of customers for profiling is selected, we need to prioritize this minimal set.

---

[40] A totally unimodular matrix is a matrix for which every square non-singular submatrix is unimodular (i.e., with determinant +1 or -1).

Table 15: Example. Defects' discovery

Defects	Customers			
	c_1	c_2	c_3	c_4
d_1		×		
d_2	×	×	×	
d_3	×		×	
d_4			×	×
d_5				×

5.4.2 Prioritization of Customers within the Minimal Set

In order to prioritize customers within the set of customers for profiling we propose to use the following greedy heuristics[41]. The customer prioritization heuristic greedily selects the customer with the largest number of non-covered defects. Once the customer is selected, the customer's defects are marked as covered. The process repeats itself until all the defects are covered at least once. We avoid applying this heuristic to the initial set of customers directly (skipping the BIP step described in the previous subsection) because of the sub-optimality of the heuristics [1].

5.4.2.1 Example of Prioritization of Customers within the Minimal Set of Customers for Profiling

Let us use the data from the example described in Section 5.4.1.1. The minimal set of customers for prioritization is $\{c_2, c_3, c_4\}$. "Non-covered" defects per customer are $\{\{d_1, d_2\}, \{d_2, d_3, d_4\}, \{d_4, d_5\}\}$, respectively. Since c_3 discovered the largest number of defects, we pick c_3 as the first customer to profile. We now mark defects d_2, d_3, and d_4 as "covered" and remove them from the non-"covered" list. The defects list is changed to $\{\{d_1\}, \{\emptyset\}, \{d_5\}\}$: customers c_2 and c_4 have one uncovered defect. We arbitrarily pick c_2 as the second customer for profiling and c_4 as the third one.

[41] That is heuristic making locally optimal choice at each stage.

5.5 Validation Case Study

Our experimental ground is a complex commercial software application with over 10 million lines of uncommented source code and a large customer base. To verify our technique, we selected defects for core components[42] of the software under study over a five-year period[43]. These defects were discovered by a few thousand customers.

In general, depending on a particular goal, an analyst may focus on a specific set of defects. For example, one can filter defects by:

- severity,
- relation to a specific functionality,
- specific symptoms (e.g., crash, data corruption).

5.5.1 Exploratory Analysis

To analyze defects based on the quadrants described in Section 5.3, we create a scatter plot of the total number of discovered defects versus the average number of rediscoveries per defect per customer (where each point represents a specific customer). Results are shown in Figure 30. As we can see, the quality assurance team does a good job of finding defects in frequently executed paths: there are not too many frequently rediscovered defects. Therefore, if we split Figure 30 in four quadrants symmetrically, there will be no customers in the top-right (HH) quadrant (i.e., there will be no "ideal candidates for profiling" as per Table 14 classification). We have to split the plot asymmetrically. Quadrant borders are selected manually and denoted by dotted lines in Figure 30. The data can then be described as

---

[42] Internal defects database data was validated using various data mining procedures.

[43] In order to stabilize the number of discoveries per defect, we select defects that were first discovered at least six months before this case study was performed.

- Bottom-left (LL) and Top-left (LH) quadrants: majority of the customers find a small number of defects that are rediscovered infrequently (LL) or frequently (LH) ("not interesting from a profiling perspective");
- Bottom-right quadrant (HL): a small number of customers discovers infrequently rediscovered defects ("potential candidates for profiling");
- Top-right quadrant (HH): a fraction of customers discovers frequently rediscovered defects ("ideal candidates for profiling").

Let us use our automatic procedure to select a minimal set of customers for profiling.

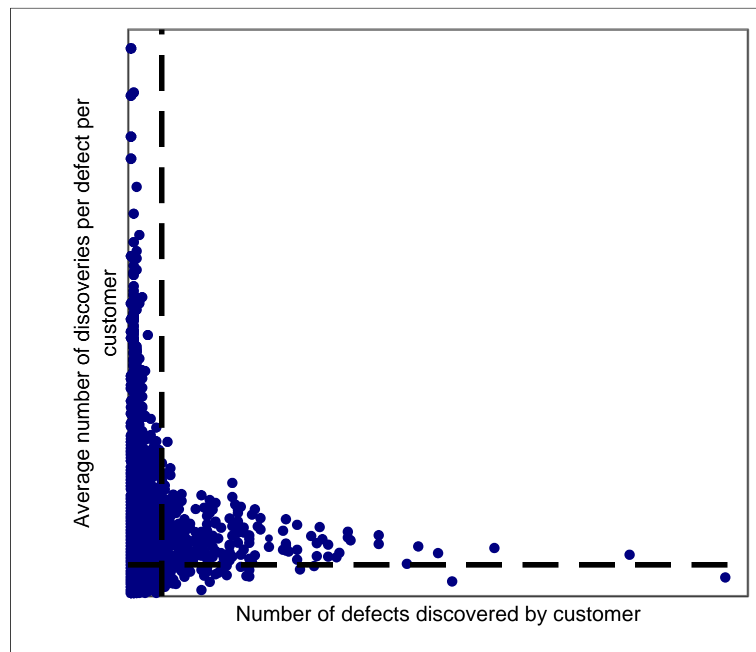

Figure 30. Total number of discovered defects vs. average number of rediscoveries per customer. Dotted lines depict borders of quadrants described in Table 14.

5.5.2 Selection of the Minimal Set of Customers

Manual selection of customers to profile is impractical. Therefore, we apply the BIP technique described in Section 5.4.1 to our dataset. We assume that all customers have equal weight: $w_i = 1$ for all i in equation (5.1). The problem is solved using IBM®

ILOG® CPLEX® solver [5]. The solution to the BIP problem is found in less than one second (on an Intel® Pentium® 4 computer) using the solver's default optimization routines.

Analysis shows that we need to profile 26% of our customers to cover all the defects found in the field. Results of customer prioritization (using greedy heuristics as described in Section 5.4.2) are shown in Figure 31 and Figure 32 ("all defects" curves) and Table 16 ("defects discovered at least 1 time" row). As we can see, the cumulative coverage curve is steep: we need to profile 9% of customers to cover 80% of defects. However, in the case of a large customer base, we may need to decrease a set of customers to profile even further. In order to do this, based on the criteria shown in

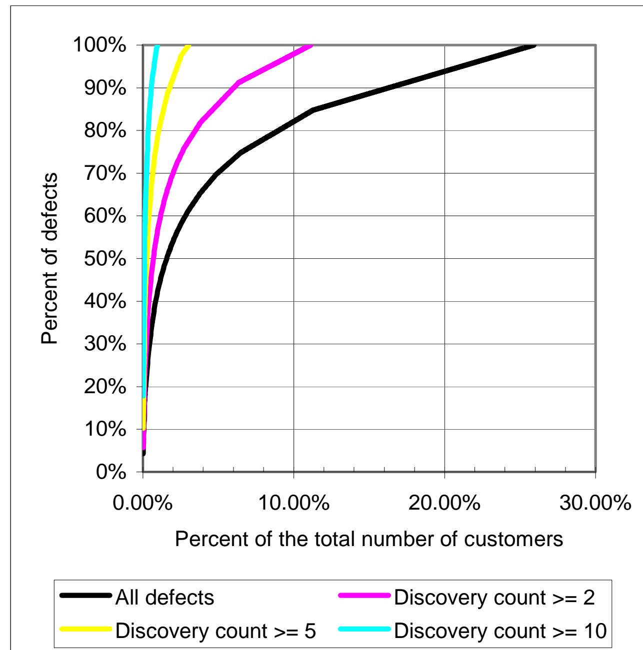

Figure 31. Percentage of the total number of customers needed to cover a certain percentage of defects of interest.

Table 14, we will focus on the defects that were discovered multiple times. Results are shown in Figure 31, Figure 32, and Table 16. The minimal set of customers decreases

rapidly as the number of defects of interest decreases. For example, we need to profile 0.9% of the customer base to cover all defects discovered at least 10 times.

Table 16. Percentage of the total number of customers needed to cover *X*% of defects discovered at least *Y* times

Defects discovered at least *Y* times	Cover *X*% of defects	
	80%	100%
1	8.9%	25.9%
2	3.5%	11.1%
5	1.1%	3.0%
10	0.4%	0.9%

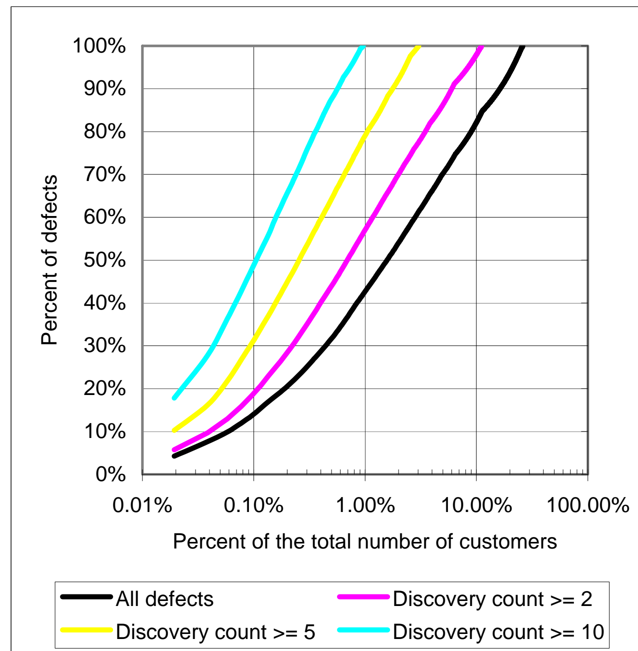

Figure 32. Percentage of the total number of customers needed to cover a certain percentage of defects of interest (log-scale).

5.6 Summary

Operational and usage profiles collected from customers provide developers and testers with valuable quantitative information on usage patterns of software being developed. Unfortunately, gathering such profiles from a large set of customers can be challenging because of time and resource constraints; moreover, customers may refuse to provide access to their systems for confidentiality and legal reasons. This limitation also leads to duplicate information being gathered, as such, information about customer defects can help us to narrow down a list of candidate customers to profile.

In this paper we discussed a technique for the selection and prioritization of a minimal set of customers for operational and usage profiling to cover a certain set of defects. This was achieved using the Binary Integer Programming algorithm. After identifying a minimal set of customers for profiling, we used greedy heuristics to prioritize the set of customers. We performed a validation case study that confirms that this approach is scalable and can produce output for a large customer base (involving thousands of customers) within seconds. In addition, we discuss defect prioritization schema based on frequency of defect rediscovery by customers. Analysis results can then be used to close gaps in testing coverage and to improve maintenance process.

References

[1] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. 2001 *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education.

[2] Elbaum, S., Rothermel, G., Kanduri, S., and Malishevsky, A. G. 2004. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Control* 12, 3 (Sep. 2004), 185-210.

[3] Gittens, M., Lutfiyya, H., and Bauer, M. 2004. An Extended Operational Profile Model. In *Proceedings of the 15th international Symposium on Software Reliability Engineering* (November 02 - 05, 2004). ISSRE. IEEE Computer Society, Washington, DC, 314-325.

[4] Hassan, A. E., Martin, D. J., Flora, P., Mansfield, P., and Dietz, D. 2008. An Industrial Case Study of Customizing Operational Profiles Using Log Compression. In *Proceedings of the 30th international Conference on Software*

Engineering (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 713-723.

[5] IBM® ILOG® CPLEX®: http://www.ilog.com/

[6] Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D. 1994. Test data generation and feasible path analysis. In *Proceedings of the 1994 ACM SIGSOFT international Symposium on Software Testing and Analysis* (Seattle, Washington, United States, August 17 - 19, 1994). T. Ostrand, Ed. ISSTA '94. ACM, New York, NY, 95-107.

[7] Musa, J. D. 1993. Operational Profiles in Software-Reliability Engineering. *IEEE Softw.* 10, 2 (Mar. 1993), 14-32.

[8] Papadimitriou, C. H. and Steiglitz, K. 1998 *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc.

[9] Weyns, K. and Runeson, P. 2007. Sensitivity of software system reliability to usage profile changes. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea, March 11 - 15, 2007). SAC '07. ACM, New York, NY, 1440-1444.

[10] Yamany, H. EL and Capretz M.A.M., 2007. A Multi-Agent Framework for Building an Automatic Operational Profile, In *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, Springer, 161-166

Chapter 6

6 Modelling Assumptions and Requirements in the Context of Project Risk

The importance of assumptions in Requirements Engineering has long been recognised. However, to the best of our knowledge, no quantitative models for the relation between assumptions and requirements are yet available. We propose a temporal mathematical model of the relationship between assumptions and requirements in the context of predicting risk associated with assumptions failure in a software project. This model incorporates two sources of structure. One, the inter-relation between assumptions and requirements are described using a Boolean network. Two, the invalidity of assumptions and the requirements change, it is assumed, may be modelled as a stochastic process. The chapter gives an illustrative example of how the model can be used to assess project risk.

6.1 Introduction

It is generally accepted among software engineers that assumptions underlie the requirements "iceberg" [12, 13] and are reflected in software. For example, the requirements for a stack of numbers could have an undocumented underlying assumption that the stack is so large on the physical device that the users could not possibly fill it up. In software, therefore, it is quite conceivable that the programmer did not test for an overflow condition. Unfortunately, the assumption can be incorrect and lead to a software failure. Many researchers thus emphasize the importance of documenting assumptions [16, 18, 13], [22, pp. 102, 157]. In our simple stack example above, the maximum allowable stack size should be made explicit, which would help in writing code to test for an overflow situation and thereby prevent software from failing at that point. Lehman and Ramil [16], in fact, even suggest that personnel have to be trained in recording and managing assumptions.

However, the validity of assumptions can change with time, for example, when the application domain or the software's context changes [3, 16, 17]. For example, if the stack software is ported to a device with a smaller possible stack size then the software can fail again if this limit is not appropriately modified upon porting. Moreover, the assumptions can be wrong from the very beginning, though the developers are not aware that they are false [7, pp. 271-272].

In practical terms, the invalidity of assumptions is a source of problems [16, 12] for software developers and users alike. For developers, for example, invalid assumptions can imply having to fix software as a consequence of software failure or quality degradation, not to mention customer dissatisfaction, loss of market share and reputation. For the users, invalid assumptions can imply anything from poor software services to increased cost of business operations because of software failures.

Thus, during software modification the *validity* of the old assumptions need to be rechecked, not only the *correctness* of "old code" as generally done during regression testing. Also, developers need to ensure that the new assumptions do not violate old ones [7, pp. 271-272], [14, 16] and if they do then the conflicts need to be resolved. All of this suggests that assumptions need to be explicitly recorded and managed, and changes to them predicted and tracked.

For an *operational* system, the volatility[44] of the validity of the assumptions can imply *shocks* to the associated implemented requirements which imply, at best, a diminishing value of the existing software system and, at worst, software failure with corresponding consequences to the end user. For a software project in the *planning* or *development*

---

[44]It is not only assumptions that are volatile. Requirements, independent of the underlying assumptions, can become "invalid", say, because the stakeholders need different services from the software over time. In this paper, we simply treat any change in requirements as the removal of old ones followed by the insertion of appropriate new ones.

stages, such volatility translates into an *invalidity risk*. The risk here is that the software being developed (or changed) may not be as desirable upon completion as first imagined. It is thus important to be able to predict, during the early stages of requirements engineering and periodically from then on during development, the amount of invalidity-risk inherent in the software project. Just note that in a software project there are also other kinds of risks to contend with, such as technical risks, personnel risk, budgetary risk, timely deliverability risk, business risk, etc., which are out of scope of this paper[45].

For the prediction of invalidity risk, there is a need to model the relation between assumptions and requirements and, using this relation, compute a measure of risk. The key idea is that if an assumption becomes invalid, it may reduce the validity of the associated requirements, thereby increasing risk. But, of course, there are assumption-assumption and requirement-requirement relationships, which must also be considered in the model. The paper defines specific metrics which serve to predict risk.

To put such a model into practice, we need to consider at least two scenarios. One is *intra-release cycle-time*, where invalidity risk is predicted at the start of the project for different time-stamps within the release cycle until the project-end. This would give us intra-release risk trends. The second scenario is prediction over *multiple releases* to obtain a risk trend over a longer period of time. The chapter describes an algorithm to cover both of these scenarios and gives an example (from a banking application) of how the model could apply in practice.

The next section describes related work. This is followed by the requirement-assumptions relationship in Section 6.3. Section 6.4 describes the modelling tools: boolean network and stochastic processes. Section 6.5 then describes the properties of requirements, the

---

[45] Therefore, unless indicated otherwise, from here on in, "risk" is meant to mean invalidity risk.

risk metrics based on these properties, and how to model risk trends. Section 6.6 gives an example simulation from a banking application. Section 6.7 then concludes the chapter.

6.2 Related work

The subject of assumptions in software systems is not new by any means. As early as the late 1960's, Lehman studied the growth complexity of the OS/360 operating system and had made growth predictions based on certain assumptions about the development processes that would be used [15]. More recently, together with Ramil [16], Lehman has explored assumptions more deeply in the context of software evolution, especially: domain changes and their impact on assumptions, mapping between assumptions and software elements, relationships with other entities of interest (e.g., economic and societal factors), need for documentation and review, a program's impact on the operational domain, management of assumptions, and so on. Also, many other authors, as described in the introduction, have referred to assumptions in their work.

It is not all theoretical however. In practice, developers make (explicit or implicit) assumptions throughout a software project though there is little computational use of assumptions in tools that could aid in achieving some tangible project goals, such as time to delivery, development within budget and quality upon delivery.

Based on some meta-models in requirements engineering [19] in which the entity assumption is related to other entities such as requirement and rationale, requirements traceability tools, such as Doors [21], Rational Suite AnalystStudio [9], and CORE [2] have been developed. While such tools allow representation of project items and traceability using inter-relationships according to the meta-model followed, they are mainly documentation and report generating tools as opposed to development or analysis tools.

In the research community, there are goal-oriented requirements engineering approaches and tools [11] which model the assumptions. The general objective is to derive a consistent and valid set of requirements for further system development. The interest in

the subject of assumptions, in this community, has been high enough to attract a conference panel session dedicated to this topic [8]. Besides giving motivation for assumptions, Greenspan raised some important questions for this panel session, such as: Who needs to keep track of the assumptions? How do we elicit assumptions? Whether there would be any immediate benefits of doing so? How can we record and manage the information? How do we use it? How much of the reasoning can be done by tools?

One of the concerns with the work on assumptions, however, is that developers are reluctant to put time and resources into documenting assumptions because the payback cycles can be long and, often, not to the person who originally documented the assumptions. For example, the assumptions underlying a requirement can be quite useful in questioning the validity of the requirement long after it has been implemented, so here, the payback is much later, possibly to a new person on the job.

One way to overcome this resistance, which we learned from our industrial collaboration, is external or internal legislation which would require that assumptions (and their rationale) be documented. Thus, in legal situations, there would be traceability of the decisions made. This is an organizational factor which also does not lend towards a concrete project goals but is usually justified in terms of business requirements.

Thus, there is a need to find ways to make short-term use of assumptions with demonstrable project benefits. The goal of our work is precisely in this direction. Operationalising our proposed model would lead to tangible results in terms of determining system invalidity-risk in different contexts. For example, when considering alternative strategies for providing a superior solution to a user, our model could help in determining the relative levels of system invalidity. Also, as a project progresses, it is important to be able to determine periodically the level of future risk perceived at that time so that corrective action can be taken as early as possible. The proposed model is thus an important aid to management decisions in software projects.

6.3 Requirements & Assumptions

Let us now formalize the assumptions properties, discussed in Section 6.2.

6.3.1 Assumptions Formalization

There exists a finite set of assumptions A_C, which completely describes the system. Elements in A_C are assumed to be atomic, i.e., if the assumption is non-atomic, then it can be represented as a larger set of simpler assumptions. As stated in [16], assumptions can be explicit or implicit, conscious or unconscious. We can quantitatively measure only documented assumptions. However, it is almost impossible to document all assumptions in A_C (see [16] and [7, p. 275]), since there is evidence that typical software projects embed at least one assumption per ten lines of code [14]. For this model we assume that the captured assumption set depicts the fundamental properties of the system.

We state that we will be able to capture a finite subset of assumptions A, such that $A \subset A_C$, depicting the main properties of the software project. The number of assumptions in A is given by N_A (the count starts from one).

Let us introduce the binary variable $V_{(\cdot)}(j,t)$, having two states

$$V_{(\cdot)}(j,t) = \begin{cases} 1, & \text{if } j-\text{th member of } (\cdot) \text{ is valid at time } t \\ 0, & \text{if } j-\text{th member of } (\cdot) \text{ is invalid at time } t \end{cases}, \qquad (6.1)$$

where $(\cdot)$ represents some set (not necessary a set of assumptions) and V returns the validity state of j-th member of the set, current time is denoted by t.

The time t validity of the j-th assumption is then given by $V_A(j,t)$ and may be in two states – valid (1) or invalid (0), for $j = 1,...,N_A$. We assume that the switching process is one-way, i.e. once the assumption becomes invalid it cannot become valid again.

Assumptions may depend on other assumptions in the set. Let us denote a dependent or "child" assumption as a_α and the set of parent assumptions as A_p. If all assumptions in A_p fail, then a_α fails too[46]:

$$\bigvee_{j=1}^{N_{A_p}} V_{A_p}(j,t) = 0 \rightarrow V_A(\alpha,t) = 0, \qquad (6.2)$$

where $\vee$ is the logical "or" and N_{A_p} is the number of elements in A_p.

A_p can be divided into two disjoint subsets: the standard assumptions A_{std} and the key assumptions A_{key}, $A_p = A_{std} \cup A_{key}$. If at least one assumption from A_{key} fails, then so does a_α:

$$\bigwedge_{j=1}^{N_{A_{key}}} V_{A_{key}}(j,t) = 0 \rightarrow V_A(\alpha,t) = 0, \qquad (6.3)$$

where $\wedge$ is the logical "and", $N_{A_{key}}$ is the number of elements in A_{key}.

If all assumptions in A_{std} fail, but at least one assumption in A_{key} is valid, it does not imply the failure of a_α:

$$\bigvee_{j=1}^{N_{A_{std}}} V_{A_{std}}(j,t) = 0 \nrightarrow V_A(\alpha,t) = 0, \text{ if } \bigwedge_{j=1}^{N_{A_{key}}} V_{A_{key}}(j,t) = 1, \qquad (6.4)$$

---

[46] A $\rightarrow$ B means if A is true then B is also true.

where $N_{A_{std}}$ is the number of elements in A_{std}. Although the failure of all assumptions in A_{std} does not imply the failure of a_α, this event could affect the probability of future survival of a_α. This will become evident when we discuss stochastic models for failures in Section 6.4.2. These relations should be specified by the user for each particular case. Note that when no key assumptions are present Equation (6.4) transforms to Equation (6.2).

6.3.2 Requirements Formalization

As in the case of assumptions, we have a finite set of requirements R_C. We are capable of capturing a finite subset of requirements R having N_R elements. A requirement in our model has a value of '1' or '0'. A '1' at any given time-state implies that the requirement is desirable (above some threshold). A '0' at any given time-state implies that either the importance of the valid requirement is below a certain threshold and, hence, is not desirable; or that the requirement is not valid. Both of these types of '0' state can induce change at the appropriate future time thereby increasing invalidity risk. However, we will still apply (6.1) in the sense that the term “valid” (“invalid”) is interpreted as “desirable” (“undesirable”).

The j-th requirement is given by the binary variable $V_R(j,t)$. As with assumptions, once a requirement is removed from specification, it cannot be re-inserted there in the future. The removal of a requirement in the specification list may lead to modification or removal of other requirements. Similar to assumptions, we postulate a dependent requirement r_β and the set of parent requirements R_p. Let the parent set be further divided into the standard R_{std} and the key R_{key} disjoint subsets of requirements, $R_p = R_{std} \cup R_{key}$.

In contrast with the assumptions model, the removal of all requirements in R_p will not necessarily (if R_{key} is empty) lead to removal of r_β from R:

$$\bigvee_{j=1}^{N_{R_p}} V_{R_p}(j,t) = 0 \nrightarrow V_R(\beta,t) = 0, \qquad (6.5)$$

where N_{R_p} is the number of elements in R_p.

A removal of a single requirement in R_{key} leads to the removal of r_β :

$$\bigwedge_{j=1}^{N_{R_{key}}} V_{R_{key}}(j,t) = 0 \rightarrow V_R(\beta,t) = 0, \qquad (6.6)$$

where $N_{R_{key}}$ is the number of elements in R_{key}.

The removal of all requirements in R_{std} does not imply the removal of r_β:

$$\bigvee_{j=1}^{N_{R_{std}}} V_{R_{std}}(j,t) = 0 \nrightarrow V_R(\beta,t) = 0, \qquad (6.7)$$

where $N_{R_{std}}$ is the number of elements in R_{std}, but, as in the assumptions case, may influence the probability of removal of r_β.Let us now consider how the assumptions influence requirements.

6.3.3 Requirements & Assumptions Interaction

In Sections 6.3.1 and 6.3.2 we treated assumptions and requirements independently. However, we know that assumptions influence requirements. We extend the ideas in the previous section and say that requirement r_β will depend not only on a parent set of requirements R_p but also on a set of underlying assumptions A_p split into A_{std} and A_{key}. Thus we postulate that the failure of all underlying assumptions or at least one key assumption will lead to "undesirability" of r_β; the failure of all assumptions in A_{std} will not lead to "undesirability" of r_β, given that at least one assumption in A_{key} is valid:

$$\begin{aligned}
&\bigvee_{j=1}^{N_{A_p}} V_{A_p}(j,t)=0 &&\rightarrow V_R(\beta,t)=0,\\
&\bigwedge_{j=1}^{N_{A_{key}}} V_{A_{key}}(j,t)=0 &&\rightarrow V_R(\beta,t)=0,\\
&\bigvee_{j=1}^{N_{A_{std}}} V_{A_{std}}(j,t)=0 &&\nrightarrow V_R(\beta,t)=0,\ \text{if}\ \bigwedge_{j=1}^{N_{A_{key}}} V_{A_{key}}(j,t)=1.
\end{aligned} \qquad (6.8)$$

Let us now consider the mathematical tools suitable for modelling this behavior.

6.4 Modelling tools

The state change of assumptions and requirements happens for various reasons, such as

- An assumption or requirement was elicited incorrectly.
- The operational domain changes which, in turn, leads to changes in the assumptions and requirements sets.
- An assumption (or requirement) changes state because parent assumption(s) (or requirement(s)) changes state.

We can think of the first two points as an “external force” acting on the system. The third point can be treated as an “internal force”, since once the relations between the members of the set have been identified, the system becomes closed -- member states of a given set depend only on the state of the parent set members. Let us first discuss an approach to modelling the “internal force” through the use of boolean networks. This is followed by a description on modelling the “external force” by an event arrival process. We then synthesize the two models into a hybrid model to show, algorithmically, how the model iterates through the time-stamps within the cycle-time for one release or through multiple cycle-times in the case of evolutionary releases. The purpose of such modelling is so that

later we can use these models to assess project risks, for example, assumptions and/or requirements change.

6.4.1 Boolean network

For modelling the dependencies between child and parent members we suggest a Boolean networks approach (see, for e.g., [10, pp. 182--203]). Boolean networks have many applications and are widely used in modelling different cybernetic and neural networks, molecular components of immune systems, etc. The network is constructed from "on-off" nodes that can take only binary values. The system's behavior[47] is governed by a set of switching rules, which are called Boolean functions. Consider the following example.

Example 6.4.1 Let us consider the toy model inspired by the study of code decay in the telephone switching systems [4]. The authors say that "...many of the original system abstractions assume that subscriber phones remain in fixed locations". Let this assumption be represented by a_α. In turn, a_α may depend on three other assumptions: a_1 – the customer does not need the roaming feature (for stationary phones); a_2 – the hardware does not support roaming; a_3 – no cell phones exists.

The relation between the above assumptions may be quite complicated. However, for pedagogical purposes, let us consider two simple configurations.

1. Configuration I. The a_α will be valid until all three parent assumptions fail. We can write the Boolean function as $[V_A(1,t) \vee V_A(2,t) \vee V_A(3,t)] = 0 \rightarrow V_A(\alpha,t) = 0$, the graphical representation is given in Figure 33.a.

---

[47] System behavior is in fact a sequence of system states at different time-stamps of interest, and system state is defined by the validity of the assumptions and requirements at any given time.

2. Configuration II. Assume that a_3 is the key assumption. Thus, if it fails then a_α fails too, even if a_1 or a_2 are still valid, see Figure 33.b. However, as in Configuration I, if only a_1 and a_2 fail then a_α is still valid. The Boolean function is given by $V_A(3,t)=0 \to V_A(\alpha,t)=0$,

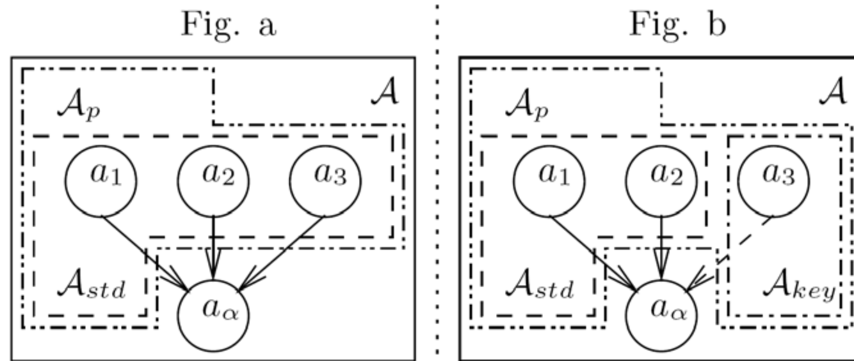

Figure 33. Example 4.1. Set up of assumptions for a. Configuration I; b. Configuration II. Solid arrows denote standard relationship, dotted arrows denote key relationship.

We may check how the Boolean functions affect the system state. In Table 17 we show how the current state of nodes at time T will affect the Boolean function at the next time instant $T+dt$, where dt is an infinitesimal time increment (we assume that the changes happen immediately).

6.4.2 Modelling Event Arrival

There are three key aspects of event modelling. One, at initial time the Boolean network is *initialized* with validity values at each node. Two, each requirement has a degree of *importance*, which can change over time. Three, the *validity* of each requirement can change over time. This section describes how this is accomplished.

Table 17. Example 4.1. State changes of assumptions.

T				$T+dt$	
Configuration				Configuration	
I & II				I	II
a_α	a_1	a_2	a_3	a_α	a_α
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	0
1	0	1	1	1	1
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	1	1	1

6.4.2.1 Modelling Incorrect Elicitation

As mentioned above, an assumption or requirement may be invalid (i.e, in state zero), even at initial time t_0, perhaps without knowing it. This can be captured by initializing the values in the network randomly, using the random draw from some statistical distribution. For instance, the binomial distribution is well suited for this type of problem. The probability of incorrect elicitation may be determined based on historical data and/or expert knowledge.

As time goes by, the operational domain and user expectations change. This may lead to assumptions failure and requirements modification or removal. There are two sources of problems that may lead to this event. The first one comes from the fact that importance of requirement changes with time and the decrease of the importance value below a certain threshold may lead to removal or change of the requirement. The second one comes from the fact that validity of an assumption or requirement can change with time.

6.4.2.2 Modelling Requirement Importance

Let us denote the time t importance of j-th requirement as $I(j,t)$. In general, $I(j,t)$ should be modeled as a stochastic process (in the simplest case it can degenerate to a constant value), since it is in general impossible to specify the importance value at some

future time instance. The parameters for this process and the value of the threshold $I_\tau(j)$ for j-th requirement should be obtained from the stakeholder. Let us consider an example.

Example 6.4.2 Suppose that we elicited requirement r and the stakeholders told us that the current importance is equal to four out of ten. They expect the importance of this requirement to grow by two units per year and the variance of this prognosis is equal to three units per year. They also mentioned that if a requirement importance drops below two units it will be removed from the specification. Let us assume that we may model the dynamics of $I(r,t)$ by a stochastic processes, to be concrete, consider here a Brownian motion [20, pp. 601-638]

$$dI(r,t) = \underbrace{\mu dt}_{\text{Deterministic}} + \underbrace{\sigma dW(t)}_{\text{Random}}, \tag{6.9}$$

where μ and σ are constants, and $W(t)$ is a Wiener process [20, pp. 601-638]. We can interpret μ as the velocity of the deterministic drift and σ captures the power of the random diffusion component. It turns out (see [20, pp. 601-638] for details) that the conditional probability distribution of importance at time $t+dt$, given the importance value at time t is normal with mean $I(r,t)+\mu dt$ and a variance $\sigma^2 dt$. In our case $\mu = 2$ and $\sigma = \sqrt{3}$. An example of the five realizations of $I(r,t)$ is given in Figure 34. As we can see, even though we expect $I(r,t)$ to grow, there is still some chance that the requirement will be removed from specification.

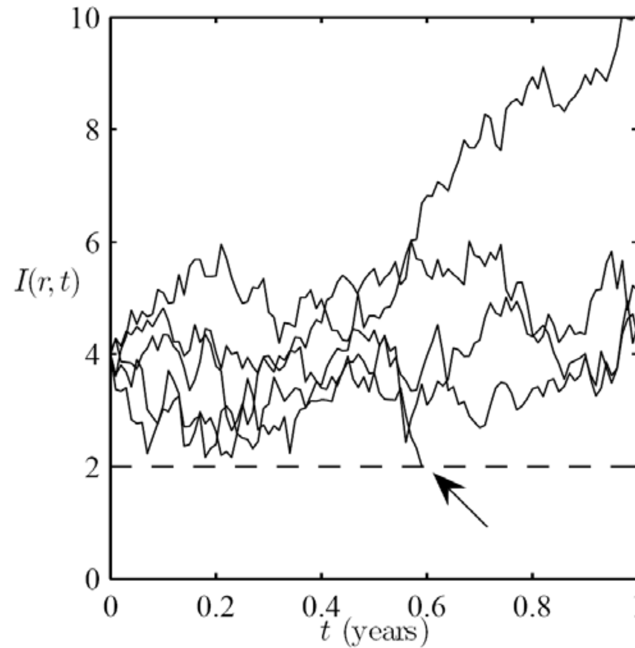

Figure 34. Example 4.2. Five random realizations of $I(r,t)$

6.4.2.3 Modelling Validity Change

The evolution in time of the Equation (6.1) for requirements and assumptions can be naturally modeled by some event arrival process. The family of Poisson processes are used to model real and discretely countable events. For our purposes we are interested in the time of the first event arrival triggering a state change at j-th node. A Poisson process is governed by an *intensity function* $\lambda(j,t)$. We can think of $\lambda(j,t)$ as the average number of events arriving per unit time. Depending on the functional form of $\lambda(j,t)$ the processes have different names: if $\lambda(j,t)$ is constant --- a Poisson process; if $\lambda(j,t)$ is a deterministic function of time --- an Inhomogeneous Poisson process; and if $\lambda(j,t)$ is governed by stochastic process --- a Doubly Stochastic Poisson process or Cox process. For a detailed discussion see, for e.g., [20, pp. 288-327] and [1, pp. 72-82, 134].

The intensity of the process may be defined by interviewing stakeholders on their opinion about the probability (or intensity) of failure of assumption or requirement at some future date. Based on this data, we may decide which process is suitable for each particular case.

The relation between the time t probability of failure of the j-th node, denoted by $P[V_{(\cdot)}(j,t)=0]$, and intensity is given by

$$P[V_{(\cdot)}(j,t)=0]=1-E\left\{\exp\left[-\int_{t_0}^{t}\lambda(j,s)ds\right]\right\}, \qquad (6.10)$$

where $E[\cdot]$ is the expectation operator. As an example, let us consider the probability of failure behavior governed by a Poisson process.

Example 4.3 For a Poisson process with constant intensity $\lambda(j)\equiv\lambda(j,t)$ Equation (6.10) simplifies to

$$P[V_{(\cdot)}(j,t)=0]=1-\exp\left\{-\lambda(j)\left(t-t_0\right)\right\}.$$

6.5 Predicting risk at time *t*

6.5.1 Risk metrics

Let us first introduce the following metrics.

- **Validity.** Time *t* validity $V_R(k,t)$ of the k-th requirement defined by Equation (6.1) is also used as a risk metric.

- **Importance.** Time *t* importance $I(k,t)$ of the k-th requirement, introduced in Section 6.4.2.2 is also used as a risk metric.

- **Children weight.** Requirements may depend on other requirements --- failure of one requirement may lead to the failure of another. Therefore, the more children a

given requirement has, the more important it is. In order to capture this property, we introduce the time t children weight $C(k,t)$ of the k-th requirement:

$$C(k,t)=\begin{cases}\dfrac{c(k,t)}{\sum_{j=1}^{N_R}c(j,t)}, \sum_{j=1}^{N_R}c(j,t)\neq 0\\ 0, \sum_{j=1}^{N_R}c(j,t)=0\end{cases}, \tag{6.11}$$

where $c(k,t)$ is the overall number of children of the k-th requirement, and the denominator is used for standardization.

- **Use-cases participation weight.** One requirement may participate in more than one use-case. The more use-cases it belongs to at a particular time, the more weight it has. The use-case weight of the k-th requirement is defined as

$$U(k,t)=\frac{\sum_{i=1}^{N_U(k,t)}\dfrac{1}{m(i,t)}}{\sum_{l=1}^{N_R}\sum_{j=1}^{N_U(l,t)}\dfrac{1}{m(j,t)}}, \tag{6.12}$$

where $m(i,t)$ is the time t number of requirements in the i-th use-case, $N_U(k,t)$ is the time t number of use-cases in which the k-th requirement participates, and the denominator is used for standardization.

Naturally, a user can collect additional properties of requirements and construct other measures that might be more suitable for her needs. Also, it is not clear at this time whether the measures based on the above properties can be aggregated into a combined measure. For this reason, the invalidity risk is predicted in the form of the n-tuple, denoted by M, and composed from the measures: validity (V_R), importance (I), children weight (C), and use-case participation weight (U):

$$M(k,t)=\{V_R(k,t), I(k,t), C(k,t), U(k,t)\}. \tag{6.13}$$

For a set of requirements we can obtain a single value by summing up the values for each of the metrics for all the requirements in the set. For example, for the set of requirements R of size N_R the total time *t* metric is given by $M(R,t) = \sum_{j=1}^{N_R} M(j,t)$.

6.5.2 Single-run Algorithm: System State at Final Time

Recall that "system state" defines how valid the system is at a given time. We merge the two types of models discussed in Sections 6.4.1 and 6.4.2 in order to compute the system state starting from the initial time to some final time in one simulation. The steps needed for this purpose are summarized in the following pseudo-algorithm.

Suppose the initial time is t_0 and we want to simulate until time T_f with time step Δt. We have at least two scenarios. One, intra-release cycle-time, where T_f is the release date for the software system and Δt is periodic assessment of the validity of the assumptions and requirements, say, based on stakeholder information. Two, over multiple releases, where T_f is some distant date of interest and Δt is release-to-release dates.

1. Set the current time $t_i = t_0$.

    a. Initialize the Boolean network and define Boolean functions and intensities of the event processes for each node.

    b. Initialize the system with random values based on the stakeholders opinion of the probability of incorrect elicitation of assumptions or requirements.

    c. Execute Boolean functions to determine the effect of validity changes.

    d. Modify the intensities of event arrival for the nodes that were affected, but have not changed to the zero state (effect of parent assumptions from A_{std} and R_{std} specified by the user).

2. While $t_i \leq T_f$

    a. Set the time $t_i = t_{i-1} + \Delta t$.

    b. For each node j where $V_{(\cdot)}(j, t_i) = 1$

    c. Determine the time of switching, T_e, as the time of first event arrival of the associated Poisson-type arrival process.

        i. If node is an assumption then

            1. If $T_e < t_i$ set $V_{(\cdot)}(j, t_i) = 0$.

        ii. If node is a requirement then

            1. Determine the value of $I(j, t_i)$

            2. If $T_e < t_i$ or $I(j, t_i) < I_\tau(j)$ set $V_{(\cdot)}(j, t_i) = 0$.

    d. Do steps 1.c and 1.d.

The result of executing this algorithm is the state of the system in terms of the validity of each requirement and assumption nodes at some final time T_f. Note that essentially we have executed the algorithm only once from t_0 to T_f. This gives us only one realization (simulation run) of the system state at time T_f. The prediction from one realization is clearly not representative. We are actually interested in the expected value of the prediction for all possible realizations of system evolution by taking an average of multiple simulation runs. This is the subject of the next section.

6.5.3 Multiple-runs Algorithm: System State at Final Time

Because of the randomness built into this we cannot simulate all possible realizations of the system. For this kind of problem we can apply Monte Carlo techniques, see [6]. The Law of Large numbers tells us [6] that for sufficiently large numbers of realization the expected value can be approximated by the average values of the n-tuple metric at time T_f obtained from different runs:

$$\widehat{M}(k,T_f) = \frac{1}{L}\sum_{n=1}^{L} M_n(k,T_f) = \{\hat{V}_R(k,t), \hat{I}(k,t), \hat{C}(k,t), \hat{U}(k,t)\}$$
$$= \begin{Bmatrix} \frac{1}{L}\sum_{n=1}^{L} V_{Rn}(k,T_f), \frac{1}{L}\sum_{n=1}^{L} I_n(k,T_f) \\ \frac{1}{L}\sum_{n=1}^{L} C_n(k,T_f), \frac{1}{L}\sum_{n=1}^{L} U_n(k,T_f) \end{Bmatrix} \tag{6.14}$$

where L is the number of system realizations, and $(\cdot)_n(k,T_f)$ is the $(\cdot)$ metric of n-th system realization at time T_f for k-th requirement.

The process can be summarized by the following pseudo-algorithm:

1. Set $sum = 0$.
2. for $n = 1$ to L
   a. Do the algorithm from Section 6.5.2 and obtain the value of $M_n(k,T_f)$.
   b. Set $sum = sum + M_n(k,T_f)$.
3. Estimator $\widehat{M}(k,T_f)$ is given by $\widehat{M}(k,T_f) = sum / L$, which is equivalent to $\widehat{M}(k,T_f) = \frac{1}{L}\sum_{n=1}^{L} M_n(k,T_f)$.

Let us now look at an example that will utilize all the mathematical tools described above.

6.6 Simulation Example

The ATM Banking system needs access to the database of bank clients. The system must be operational one year from now. Let us denote this requirement as r_1. Two groups of stakeholders gave the following requirements: we have to implement the system using a centralized database, denoted as X1 model, requirement r_2; or a distributed database denoted as X2 model, requirement r_3. Clearly, r_2 and r_3 are conflicting requirements. The stakeholders gave the following assumptions underlying r_2:

a_1 – the developers are proficient in implementing X1 model,

a_2 – the X1 will handle the heavy transaction load;

and the following assumptions for r_3:

a_3 – the developers are proficient at implementing X2 model;

a_4 – the X2 will handle the heavy transaction load.

We also add a single assumption to r_1:

a_5 – we assume that one year term given for implementation is a strict deadline.

We may also deduce that the invalidity of a_4 will imply invalidity of a_2. The failure of r_2 or r_3 will lead to the failure of r_1.

We have two methods for implementing a single use-case; which one is less risky? Let us assume that there is no relation between these sets of assumptions and requirements, and the rest of the system. Thus, we can treat the use-cases as separate systems.

Note that these use-cases are mutually exclusive. That is why during the calculation of $\hat{U}(k,t)$ we assume that there is only one use-case. We model the dynamics of $\hat{I}(j,t)$ using Brownian motion described in Example 6.4.2, using (6.9). The properties collected from the stakeholder are given in Table 18 and Table 19. Both use-cases will have r_1 in them. In order not to confuse them, let us denote the one in the X1 model as $\dot{r}_1$ and the one in the *X2 model as $\ddot{r}_1$. Both instances of requirement r_1 will have the same properties at start time. Therefore, the X1 model requirements set will be given by $R_1 = \{\dot{r}_1, r_2\}$, and X2 set by $R_2 = \{\ddot{r}_1, r_2\}$. The relations between assumptions and requirements are given in Figure 35.

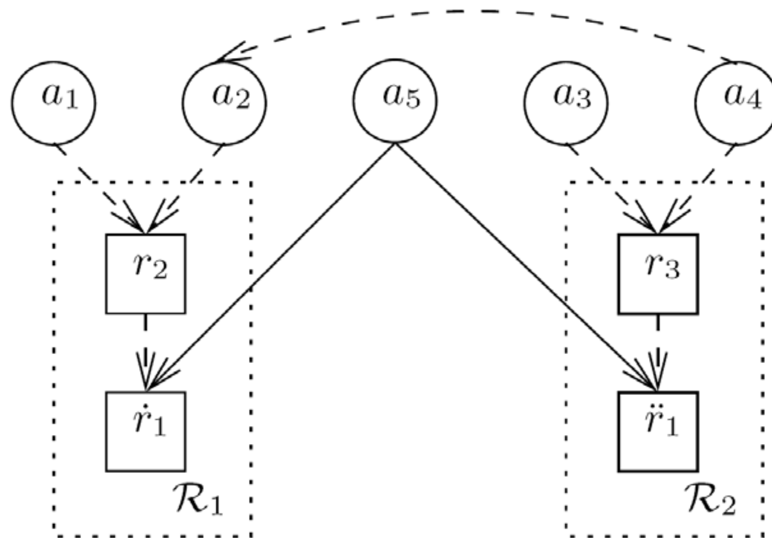

Figure 35. Simulation setup. Circles denote assumptions, squares denote requirements. Solid arrows denote standard relationship, dotted arrows denote key relationship.

Table 18. Assumptions properties

	a_1	a_2	a_3	a_4	a_5
$\lambda(\cdot,t)$	0.05	0.15	0.20	0.05	0.01

Table 19. Requirements properties

	R_1		R_2	
	$\dot{r}_1$	r_2	$\ddot{r}_1$	r_3
$\lambda(\cdot,t)$	0.01	0.02	0.01	0.02
$I(\cdot,0)$	0.60	0.40	0.60	0.40
μ	0.10	0.25	0.10	0.20
σ	0.10	0.20	0.10	0.25
$C(\cdot,0)$	0.00	1.00	0.00	1.00
$U(\cdot,0)$	0.50	0.50	0.50	0.50

We simulate the system behavior from $t_0 = 0$ until $T_f = 1$ (we assume that time is measured in years) with a weekly time step $\Delta t = 1/52$. We also say that the requirements and assumptions are elicited incorrectly with probability 0.02 (per year) and model this using a binomial distribution. We also assume that failure of any parent standard node will lead to an increase of child node intensity by 10%.

The average values of metrics for all requirements are obtained from ten thousand realizations. We re-run each system realization simulation one hundred times to obtain the standard deviation (sd) measurements. In order to obtain cumulative measures for requirements in R_1 and R_2 we sum up the metric values for each of the requirements in the use-case. The smaller the value the bigger the risk.

The metric values at $T_f = 1$ are given in Table 20. The dynamics of the metrics over time is given in Figures 4, 5, 6, and 7. From these Figures we see that values of all four metrics at the initial time were higher for the R_2 set than the R_1 set, i.e., $\widehat{M}(R_2,0) > \widehat{M}(R_1,0)$. However, at the final time, three metrics of the n-tuple $M(R_2,1)$, namely $\hat{V}_R(R_2,1)$, $\hat{C}(R_2,1)$, and $\hat{U}(R_2,1)$, are smaller than the same metrics from

$\widehat{M}(R_1,1)$. This tells us that the invalidity risk associated with implementation of model X2 would be higher than the one associated with model X1. On the other hand, the importance $\hat{I}(R_2,1)$ of requirements in R_2 is still higher than in R_1. Based on this management can decide whether to implement R_1, which has less invalidity risk, or to implement R_2, which is deemed more important at time T_f.

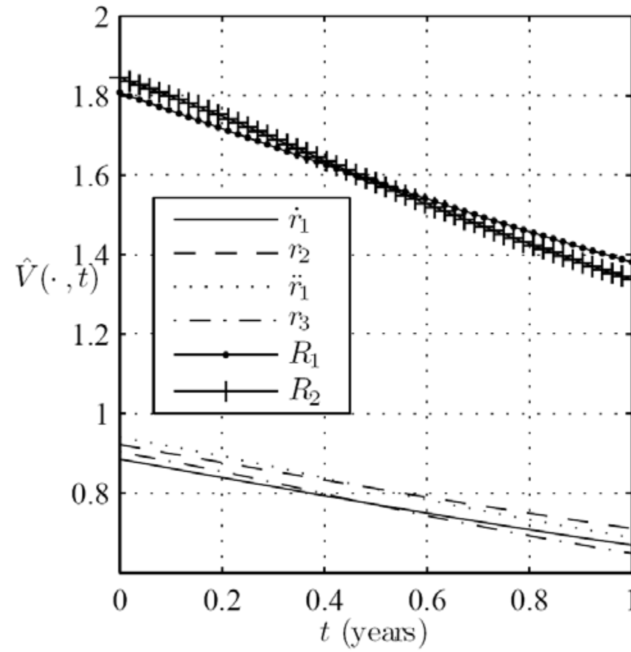

Figure 36. The value of $\hat{V}(\cdot,t)$

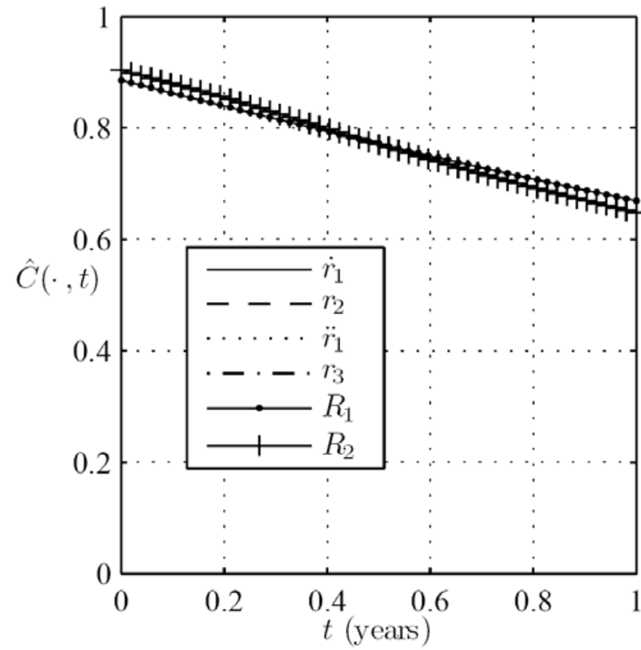

Figure 37. The value of $\hat{C}(\cdot,t)$.

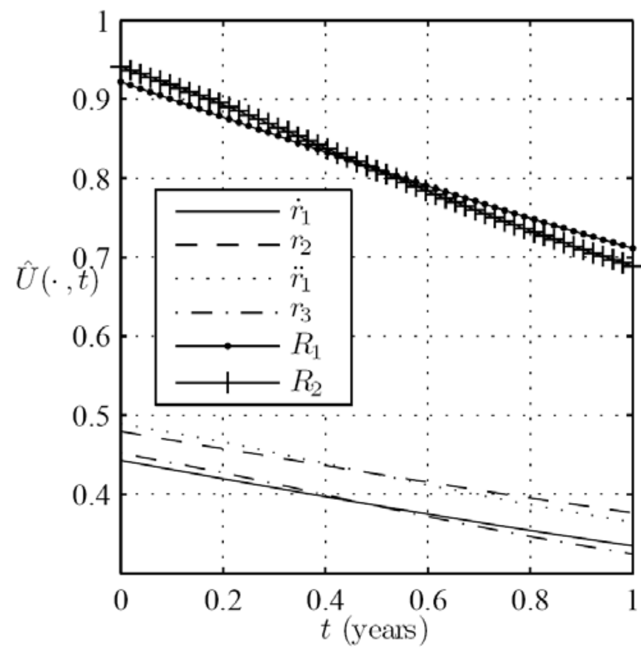

Figure 38. The value of $\hat{U}(\cdot,t)$.

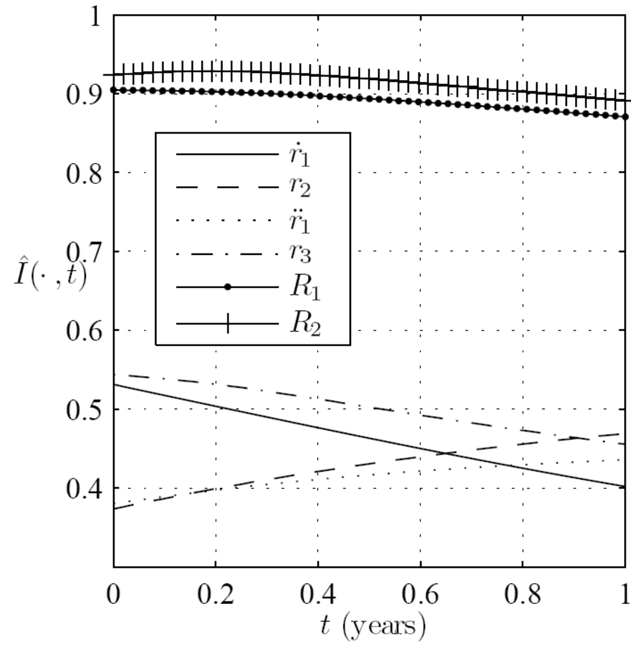

Figure 39. The value of $\hat{I}(\cdot,t)$

Table 20. Metrics values at $T_f = 1$ (± denotes standard deviation)

	$\hat{V}(\cdot,1)$	$\hat{I}(\cdot,1)$	$\hat{C}(\cdot,1)$	$\hat{U}(\cdot,1)$
$\dot{r}_1$	0.670	0.402	0.000	0.335
±	0.005	0.003	0.000	0.002
r_2	0.711	0.469	0.670	0.377
±	0.004	0.003	0.005	0.003
R_1	1.381	0.871	0.670	0.711
±	0.005	0.003	0.002	0.002
$\ddot{r}_1$	0.648	0.455	0.000	0.324
±	0.004	0.003	0.000	0.002
r_3	0.689	0.436	0.648	0.365
±	0.004	0.003	0.004	0.003
R_2	1.337	0.891	0.648	0.689
±	0.004	0.003	0.002	0.002

6.7 Conclusions & Future Work

In this paper we establish a temporal, mathematical, model which describes the interactions between assumptions and requirements of a software system in the context of predicting the system's validity risk. We capture these relations using a Boolean Network. The validity of the system over time is modeled using stochastic processes. An illustrative example from the banking domain is given. In order to perform computations we have developed a prototype software tool (not described in this paper due to lack of space).

This work cuts through the barrier solidly experienced by practitioners that documenting assumptions does not have a short-term payback [8]. In fact, it liberates them into using documented assumptions (and requirements) properties to make assessment about a system's invalidity over time (either in the intra-release context or over multiple releases context).

Voicing the concerns of numerous researchers, Finkelstein and Kramer, in [5], pose a critical question as to how to predict the effect of requirements change on a software system. In this paper, we have demonstrated a *proof of concept* that modelling assumptions and related requirements, supported by an underlying computing engine (Boolean Network and stochastic processes), it is indeed possible to predict the effect of external changes on the validity of a software system over time.

Our work in this area continues with investigation on system *usability assumptions* developers make and how they correspond to system testing amongst other aspects of software development.

References

[1] P.K. Andersen and Ø. Borgan and R.D. Gill and N. Keiding. *Statistical Models Based on Counting Processes* of Springer Series in Statistics. Springer-Verlag, New York, 1993.

[2] Vitech Corporation. *CORE*. http://www.vtcorp.com/.

[3] A.H. Dutoit and B. Paech. Rationale-Based Use Case Specification. *Requirements Engineering*, 7(1):3-19, 2002.

[4] S.G. Eick and T.L. Graves and A.F. Karr and J.S. Marron and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1-12, 2001.

[5] A. Finkelstein and J. Kramer. Software engineering: a roadmap. *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 3--22, 2000. ACM Press.

[6] G. S. Fishman. *Monte Carlo* of Springer Series in Operations Research. Springer-Verlag, New-York, 2nd edition, 1996.

[7] D.C. Gause and G.M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House Publishing Company, 1999.

[8] S. Greenspan. Panel on recording requirements assumptions and rationale. *IEEE International Symposium on Requirements Engineering*, pages 282, San Diego, 1993. IEEE Computer Society.

[9] IBM. *Rational Suite AnalystStudio*. http://www.ibm.com/.

[10] S.A. Kauffman. *The Origins of Order. Self-Organization and Selection in Evolution*. Oxford University Press, Oxford, 1993.

[11] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice . *Proc. RE'04: 12th IEEE International Requirements Engineering Conference,*, pages 4-8, Kyoto, 2004. IEEE Computer Society.

[12] A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 5--19, 2000. ACM Press.

[13] A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Trans. Softw. Eng.*, 26(10):978--1005, 2000.

[14] M.M. Lehman. Software's Future: Managing Evolution. *IEEE Software*, 15(1):40-44, 1998.

[15] M.M. Lehman. The Programming Process. IBM Research Report, RC 2722, IBM Research Centre, Yorktown Heights, NY, 1969.

[16] M.M. Lehman and J.F. Ramil. Rules and Tools for Software Evolution Planning and Management. *Ann. Softw. Eng.*, 11(1):15-44, 2001.

[17] D.L. Parnas. Software aging. *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279--287, 1994. IEEE Computer Society Press.

[18] A. Porter and L. Votta. Comparing Detection Methods For Software Requirements Inspections: A Replication Using Professional Subjects. *Empirical Software Engineering*, 3(4):355-379, 1998.

[19] B. Ramesh and M. Jarke. Toward Reference Models for Requirements Traceability. *IEEE Trans. Softw. Eng.*, 27(1):58--93, 2001.

[20] S.M. Ross. *Introduction to Probability Models*. Academic Press, San Diego, 8th edition, 2002.

[21] Telelogic. *DOORS/ERS*. http://www.telelogic.com/.

[22] K.E. Wiegers. *Software Requirements*. Microsoft Press, 2nd edition, 2003.

Chapter 7

7 Managing the Escalation of Requirements

In this chapter we describe quantitative models that can be used for managing the injection of requirements late in the software development process. We define such requirements as “escalated requirements”. The models can aid in (1) predicting escalation of previously elicited requirements, and (2) managing resources due to the escalation of completely new (unknown during elicitation) requirements.

7.1 Introduction

Deciding on the set of requirements to be implemented in a future release of a software system is done early in the development cycle, usually after the requirements prioritization phase. However, for reasons difficult to control, at times requirements get injected later in the development cycle. Typical examples would include a request from a key customer to implement a “must-have” feature in the upcoming release of the software system; a request from an executive who promised to deliver a certain feature to a client to win a contract; or newly elicited requirements that take advantage of market shifts that occur during development. We define such a situation as an “escalation” of a requirement. An *escalated requirement* (ER) is a requirement that is not in the original development plan[48]. Typically, a development plan will need to be approved by senior management and acts as the “contract” between developers and managers. The contract enforces that a committed requirement cannot be removed entirely or be substantially changed. However, the contract does not prevent new requirements from being added to development. The condition here is thus stricter for removal than for insertion.

---

[48] We define the *development plan* as a set of requirements which have to be implemented in the future release of a software system.

Note that an escalation will generally not occur in projects where the set of requirements to be implemented remains constant after the requirements engineering phase is complete, e.g., in the waterfall software development model [1]. Also, projects that utilize short development cycles (e.g. an agile development model [2]) will likely experience less of such situations because they are not locked into a long-term development plan.

The escalation of requirements may be potentially valuable from a financial (business) perspective: the implementation of an ER could have a positive revenue effect. However, it can present difficulties from the management perspective. Reactive solutions lead to schedule disruption, since possible solutions are: (1) to remove some requirements from the plan so resources can be allocated to the ER, or (2) to increase the load on the development team. These solutions inevitably affect morale and consume management's time. Software quality, project delivery date, and other factors may also be affected. Note that the later a requirement gets escalated in the development cycle, the more difficult it is to "squeeze" the requirement into the plan. The number of ERs may be fairly small, compared to the whole set of requirements in the plan. However, injection of even a single requirement can pose a challenge to the managers, depending on such factors as its implementation complexity, resource consumption, release delays, etc. Therefore, a proactive solution is desirable where requirements that would be escalated in the project are predicted suitably ahead of time and appropriate measures taken in a preventative manner.

ERs can be divided into two groups: (i) *known requirements,* i.e., those that were previously elicited but not considered of high enough priority to include in the project plan, and (ii) *unknown requirements,* i.e., those that were not known prior to their escalation and, hence, are essentially entirely new requirements that were identified during development. Based on the authors' experience, roughly 20% of escalated requirements are known requirements and 80% are unknown.

If a set of known requirements that could potentially escalate in the future were known in advance, then the managers could meticulously examine these requirements, discuss them

with stakeholders and make a decision about adding these requirements to the plan in advance. In the case of unknown requirements, an estimate of the number, size, and potential source and time of an escalation of these requirements may help managers in reserving the workforce needed to implement these requirements.

In this chapter, we propose potential approaches for forecasting the escalation of known requirements and improving resource management for the escalation of unknown requirements. This is an important practical problem, that is to the best of our knowledge, has not been addressed in the literature. Section 7.2 discusses potential solution approaches in detail, and Section 7.3 concludes the chapter.

7.2 Modeling the Escalation of Requirements

Before approving the addition of an ER to a development plan during the development cycle, managers should consider certain dimensions (or factors) associated with this requirement that will affect its prioritization. Certain dimensions, such as personal preferences, business value, implementation cost and dependencies among requirements [3], will be common to processes of prioritizing both normal and escalated requirements. However, there exist prioritization dimensions that will be specific to either normal or escalated requirements. For example, in the case of prioritization of normal requirements, managers often consider requirement stability [4], legal mandate [5], and requirement reuse. However, these dimensions are not utilized in prioritization of escalated requirements. Prioritization of escalated requirements will also have specific dimensions, such as requirements rigidity[49], schedule risk[50], and tradeoff risk[51].

---

[49] Can we implement only core features of a requirement in the current release and leave the rest for future releases?

[50] How far are we in the development phase?

[51] What should be given up to absorb a given requirement into the plan?

In Section 7.2.1 we describe a potential approach for modeling escalations of known requirements by simulating the evolution of prioritization dimensions. Section 7.2.2 considers the case of unknown escalated requirements by analyzing historical data on escalation of requirements.

7.2.1 Modeling the Escalation of Known Requirements

Independent of dimensions used for prioritization (of both normal and escalated requirements), we may depict the prioritization process (at a high level of abstraction) as a procedure for splitting a set of requirements into two groups: in or out of the development plan based on their relation to a (possibly n-dimensional) prioritization threshold P_T.

For example, we may prioritize requirements based on the complexity of integrating a requirement into the product, and the business value. The two-dimensional prioritization rule may then informally be defined as follows. If integration complexity is low and business value is higher than $1M, then this requirement should be implemented.

Before we proceed, let us establish formal notation. Without loss of generality, we may assume that the development process starts at time t_0 and ends at time t_S. Let us denote values of prioritization dimensions (or, in short, *priority*) for j-th requirement at time t as $P(j,t)$. Clearly, $P(j,t)$ may change with time. If $P(j,t)$ exceeds P_T before the end of the current development cycle, the j-th requirement gets escalated into the current release. Formally, the requirement becomes escalated if $P(j,t) > P_T$ and $t < t_S$.

In order to assess the escalation risk of known requirements we need to follow the following process

1. Tap into expert knowledge and analyze historical data on requirements escalations.
2. Identify requirements likely to cross P_T before the end of the current development cycle.

3. Once requirements that can potentially escalate are identified -- sort them by anticipated escalation-time: requirements that are expected to cross P_T early in the development cycle are less risky than those escalated late in the development cycle.
   - For the "early-crossers" (moderate cases): examine them for inclusion in the development plan.
   - For the "late-crossers" (riskier cases)[52]: during the requirements process, analyze the stakeholders' information on the likelihood of their escalation.

Note that for a sustained solution, requirement engineers need to examine the requirements process as to "why" they were not considered for the plan in the first place. If the "current" requirements process is considered satisfactory then, perhaps, it is time to question this belief.

The first and third stages of the process are relatively straightforward. In order to implement the second stage, let us examine a technique for modeling the stochastic evolution of $P(j,t)$ with time.

7.2.1.1 Modeling Evolution of Priority in Time

In order to model the evolution of a requirement's importance with time, in general, we will need the following input variables:

- Starting (current) value of the j-th requirement's priority $P(j,0)$,
- Target date values of the requirement's priority at a future time (e.g., three months, end of development, etc.) $P(j,\tau)$, $\tau > 0$,
- Dependencies that may trigger escalation of related requirements.

---

[52] If the "current" requirements process is considered "optimal" then try to find innovative ways to improve upon this "optimal" situation.

The starting value of a requirement's importance and dependencies among requirements are, usually, readily available. In order to obtain target date values we need to survey stakeholders and experts on their opinions.

There exists numerous ways of running these types of surveys. In this paper, for the sake of simplicity, we assume that at a future time instance t_F the expected priority of the j-th requirement $P(j, t_F)$ is estimated by averaging out experts' opinions. Volatility of their opinions is given by standard deviation of experts' opinions $V(j, t_F)$. Let us assume that the drift (growth trend) of requirements priority is linear[53]. We can then estimate an average drift of $P(j, t)$ per unit time as $m_j = (P(j, t_F) - P(j, t_0)) / (t_F - t_0)$.

$P(j,t)$ should be modeled as a stochastic process, since it is in general impossible to specify the priority value at some future time instance. Stochastic processes can take different forms and parameters can be determined by analyzing data.

Consider the following example. Suppose that we prioritize requirements based on a single prioritization dimension. By surveying experts, we determine that priority value of the r-th requirement $P(r, t)$ grows by two units of priority dimension per year and the variance of this prognosis is equal to three $(\text{units})^2$ per year. Let us assume that we may model the dynamics using Brownian motion [6]:

$$dP(r,t) = m_r dt + s_r dW(t),$$

where m_r and s_r are constants, and $W(t)$ is a Wiener process [6]. We can interpret m_r as the velocity of the deterministic drift and $s_r = V(r)^{1/2}$ captures the power of random diffusion component. It turns out that the conditional probability distribution of $P(r,t)$ at time $t + dt$, given the priority value at time t is normal with mean $P(r,t_0) + m_r\, dt$ and a variance $s_r^2 dt$. In our case $m_r = 2$ and $s_r = 3^{1/2}$.

---

[53] An actual form of the drift can be estimated from the collected data.

Dependencies among requirements can be modeled as follows: if the priority of a given requirement reaches P_T at time t_e, then all dependent requirements will also escalate at the same time, i.e. their priority at time t_e will also be set to P_T.

Let us put all the pieces of the model together and consider the following example. Suppose we have six known requirements (R1, R2, … , R6), with dependencies given in Figure 40. Escalation of R1 will trigger escalation of R2, R3 and R4; escalation of R5 will trigger escalation of R3 and so on. Our time horizon of interest t_S = 6 month; we perform 5000 Monte Carlo simulations with one week time steps. $P(j,t)$ is one-dimensional. Priority escalation threshold $P_T = 4$. Initial priority of requirements $P(j,t_0)$ is given in Table 21. We consider three cases with three different sets of initial Brownian motion parameters, also given in Table 21.

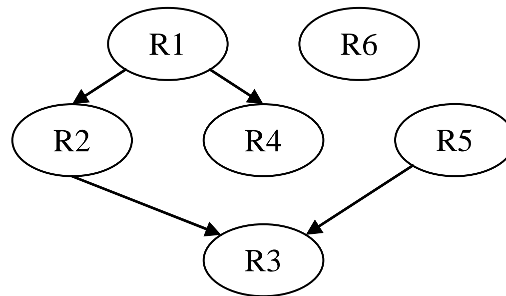

Figure 40. Dependencies among requirements

Table 21. Setup parameters

Requirement ID (j)	Initial Priority $P(j,t_0)$	Priority diffusion parameters (per year)					
		Case 1		Case 2		Case 3	
		m_j	s_j^2	m_j	s_j^2	m_j	s_j^2
R1	3.0	0.00	0.00	0.00	0.00	6.00	1.00
R2	2.5	6.00	0.00	6.00	2.00	6.00	2.00
R3	2.0	0.00	0.00	0.00	1.00	0.00	1.00
R4	1.5	0.00	0.00	0.00	0.00	2.00	0.00
R5	1.0	0.00	0.00	0.00	0.00	0.00	0.00
R6	0.5	2.00	0.00	2.00	0.00	2.00	0.00

In Case 1 the diffusion parameter s_j is equal to zero for all j -- the system shows deterministic linear growth, for those requirement where $m_j \neq 0$, and remains constant otherwise. The evolution of requirements' priority is given in Figure 41. For example, R6 grows at 2 priority units per year, and, therefore, in 6 months its priority rises from 0.5 to 1.5. R2 reaches I_T in 3 months, and this triggers escalation of requirement R3 at the same time.

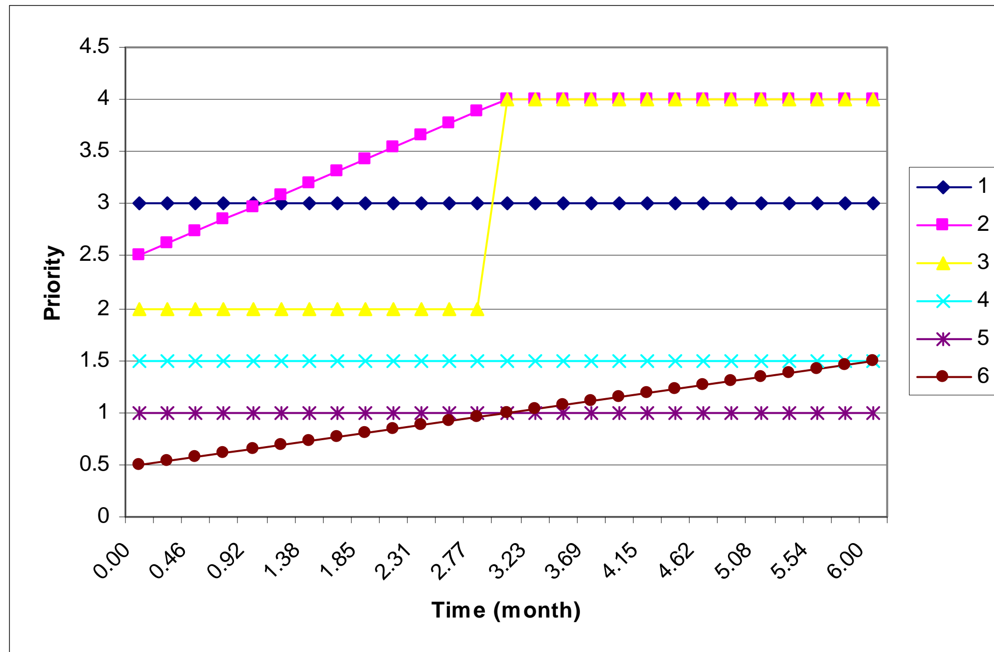

Figure 41. Case 1. Expected priority

In Case 2 we add some randomness to the model -- s_j is now non-zero for requirements R2 and R3. The expected times of escalation (see Table 22) change as randomness increase. As we can see from Figure 42, priority of R2 and R3 experience non-linear growth.

Table 22. Expected escalation time

Requirement ID (*j*)	Expected escalation time (month)		
	Case1	Case 2	Case 3
R1	-	-	2.31
R2	3.00	3.46	2.77
R3	3.00	4.85	3.46
R4	-	-	4.38
R5	-	-	-
R6	-	-	-

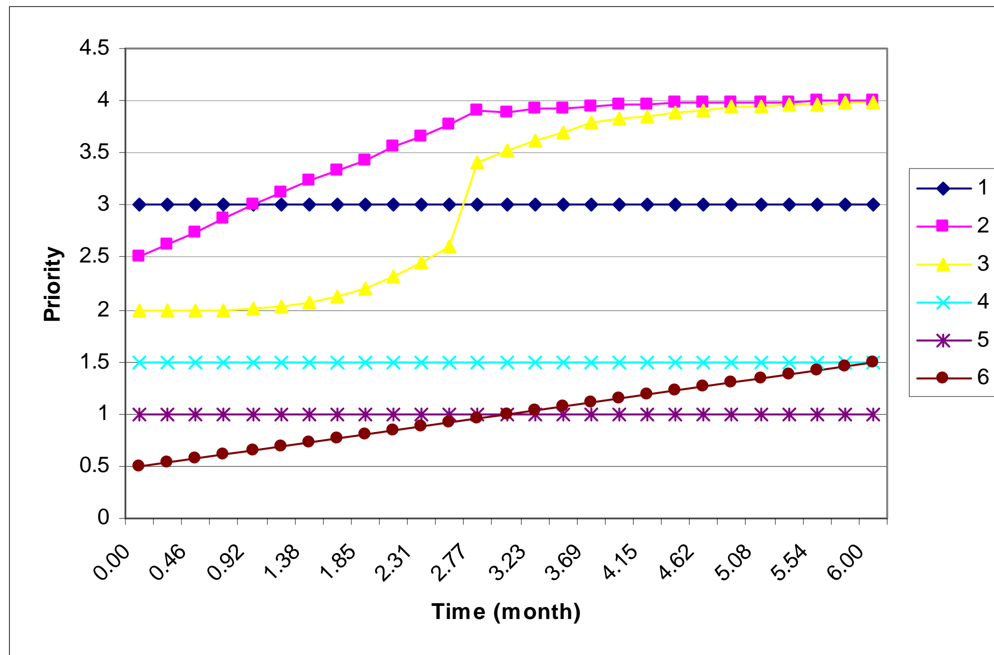

Figure 42. Case 2. Expected priority

In Case 3 we add positive drift to R1 and negative to R4. The evolution of requirements priority is given in Figure 43, and expected escalation time is given in Table 22. Note that the expected escalation time of R2 and R3 decreases due to the escalation of R1. Even though R4 has negative drift, escalation of R1 “pushes” it to escalate too.

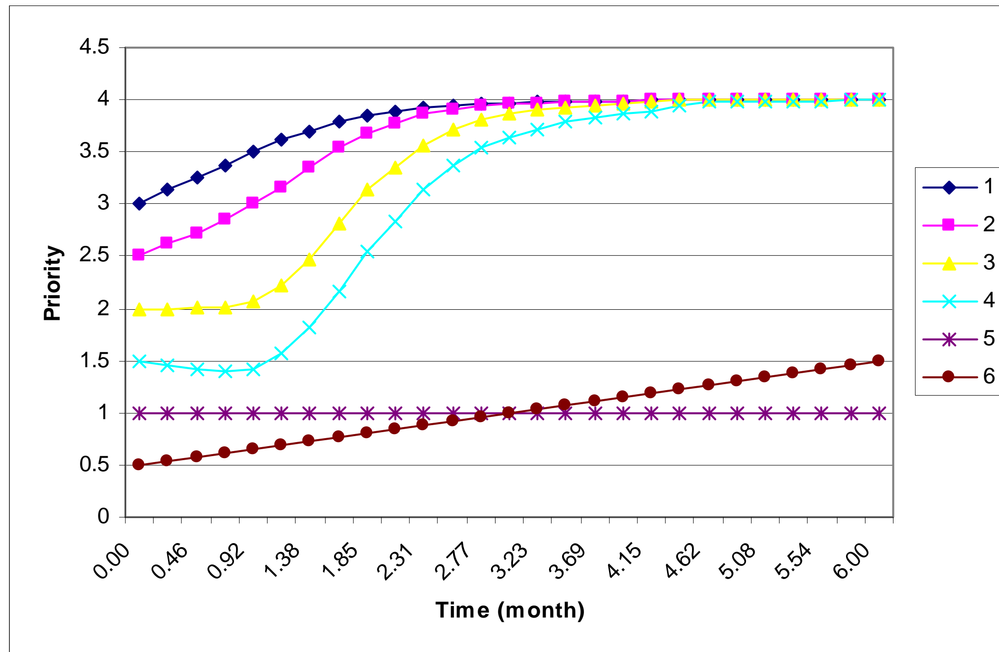

Figure 43. Case 3. Expected priority

As we can see, this simple model can capture rather complicated relationships among requirements. Once a table of expected escalation times is available, a requirements engineer can analyze the data and decide what should be done with requirements that may potentially escalate within the current development cycle.

Note that this model can be extended to multidimensional $P(j,t)$. In this case we either project different dimensions onto one, e.g., such as by taking a weighted average of different dimensions, or by simulating the evolution of different dimensions using multidimensional Brownian motion (or another stochastic process). Let us now consider the escalation of unknown requirements.

7.2.1.2 Modeling the Escalation of Unknown Requirements

Forecasting the escalation of unknown requirements can be performed by analyzing historical data on escalations. The following procedure should be performed:

1. Identify "trouble-making" clients, i.e., those that have injected requirements in the past.
2. Estimate the number and complexity of requirements the identified clients (from the step 1 above) have injected in the past[54].
3. Expected time of injection can be obtained by looking at a client's schedule – injection typically happens during (or after) their elicitation phase.

Once these data are obtained you can estimate when and what amount of resources may be needed and plan the load on development teams accordingly.

For example, if you identified that Client A injects 5±2 requirements every release and the time required to implement each of those requirements is 100±50 human hours. The client will start their next requirement elicitation 3 months from now. Your rough estimate will be as follows: you will need 5×100 = 500 human hours on average; best[55] scenario is 3×50 = 150 human/hours; and your worst case is 7×150 = 1050 human hours). You may expect that you will need this labor force not earlier than 3 month from now. A more complicated scrutiny, e.g. using time series analysis [7] can be performed if needed.

The above-described approach is most effective when the project is able to reserve development effort for unknown requirements escalations. However, based on our discussion with industrial contacts, this is not always possible. In industrial projects, often the developers will be "overloaded" with tasks. There is then a need to provide a contingency plan that is agreed to up-front for dealing with unknown requirement escalations. This contingency plan could include deliberately planning not to implement a few, lower-priority requirements until late in the development lifecycle. If and when unknown escalations occur, these low-priority requirements can be de-scoped (i.e., their

---

[54] The data may be normalized by the number of requirements implemented in previous releases of the software.

[55] The best scenario is no escalated requirements at all.

functionality reduced) or dropped altogether. The developers who were originally assigned to work on these requirements would then be free to implement the newly escalated requirements. The advantage of this contingency plan is that by planning this early in the project, there would be no “floundering” on the part of management or developers when escalations occur; the process would be set in place and understood by the developers. Also, no development effort is wasted from partially implementing requirements that are subsequently abandoned because higher-priority escalated requirements are injected into development.

7.3 Conclusions and Future Work

In this chapter we describe quantitative models that can be used for managing injection of requirements late in the software development process. In Section 7.2.1 we present a stochastic model for escalation of previously elicited requirements prediction. Section 7.2.2 describes various approaches for managing resources for the escalation of new requirements using historical data on escalations. In the future, we plan to validate these models on industrial data sets of escalated requirements.

References

1. Royce, W.: Managing the Development of Large Software Systems. Proceedings of IEEE WESCON, Vol. 26 (August) (1970) 1-9
2. Beck, K., Fowler, M.: Planning Extreme Programming. Addison-Wesley (2001)
3. Davis, A.: The Art of Requirements Triage. IEEE Computer 36, 3 (2003) 42-49
4. Fellows, L., Hooks, I.: A Case for Priority Classifying Requirements. Proceedings of 8th Annual International Symposium on Systems Engineering (1998)
5. Wiegers, K.E.: Karl Wiegers Describes 10 Requirements Traps to Avoid. Software Testing & Quality Engineering Jan-Feb (2000)
6. Ross, S.M.: Introduction to Probability Models. Academic Press, San Diego (2001)
7. Brockwell, P.J., Davis, R.A.: Time Series: Theory and Methods Springer (1991)

Chapter 8

8 Conclusions and Future Work

We started the thesis with a basic overview of the Software Engineering discipline and issues that it faces. We also described knowledge areas comprising this discipline. Next, we presented six papers grouped by Software discipline. The papers built quantitative models addressing various types of software risk. According to Hall [1], "*software risk* is a measure of the likelihood and loss of an unsatisfactory outcome affecting the software project, process, or product." *Software project risk* deals with operational, organizational and contractual aspects of the software development process. Examples of project risks are resource constraints and relations with external suppliers. *Software process risk* relates to management and technical processes. This risk is associated with activities such as planning, staffing, and quality assurance in management procedures; and requirements analysis, coding, and testing in technical procedures. *Software product risk* deals with the product characteristics. This risk arises due to requirements volatility, code complexity, incorrect test specifications, etc. Software project risk is the managers' responsibility; software product risk is handled mainly[56] by technical staff. Software process risk is mitigated by both managers and technical personnel.

In the first paper (Chapter 2) we proposed an iterative-unfolding technique for filtering a set of traces relevant to a specific task. This technique can be used to support non-scalable trace analysis tools used in software testing and maintenance leading to improved product quality and faster problem determination. We also presented a validation case study, based on industrial data, proving scalability of this approach.

---

[56] Product managers are also responsible for handling some aspects of product risk, such as requirements volatility.

In the second paper (Chapter 3) we analyzed the applicability both of the Shannon entropy and the three extended entropies (Landsberg-Vedral, Rényi, and Tsallis) to the predictive classification of traces (either stand-alone or as part of the iterative-unfolding process) described in the previous paper. Our validating case study showed promising performance of the extended entropies for classification task.

The first and second papers, dealt with the product and process risks. The techniques that we proposed in these papers can speed up problem determination of defects encountered by customers, leading to a decrease of lost opportunities by increasing customer satisfaction (due to faster problem resolution). Faster problem resolution also leads to easing of resource constraints, decreasing project risk.

In the third paper (Chapter 4) we used mathematical tools such as heavy-tail Kappa distribution and G/M/k queuing model to develop a set of metrics helping to identify gaps in quality assurance processes (addressing process risk), to allocate resources of service and maintenance teams (decreasing project and process risks), and to help customers to asses risk associated with usage of a given software product (improving customer relations and opening new opportunities). We validated the metrics using industrial data.

In the fourth paper (Chapter 5) we proposed a technique for selection and prioritization of a minimal set of customers for profiling. The minimal set of customers has been identified using Binary Integer Programming algorithm; this set was later prioritized using a greedy heuristics. We also presented a validation case study, based on industrial data, showing that this approach is scalable and can produce usable results for a product with a large customer base. Input data from customers identified using this algorithm should improve code coverage by targeting problematic functionality frequently utilized by the users, leading to improved test specifications (decrease of product risk). Analysis of customer workloads also helps analysts to better comprehend user behavior, resulting in clearer quality assurance policies with a concomitant decrease of the process risk.

In the fifth paper (Chapter 6) we established a model combining Boolean networks and stochastic processes. This model described the interaction between requirements and underlying assumptions in the context of system validity. This proof-of-concept model simulated the effect of external changes on the validity of a software system over time.

The thesis is concluded with the sixth paper (Chapter 7), where we described quantitative models that simulated the injection of requirements late in the software development process. We presented a stochastic model for prediction of escalation of previously elicited requirements and described approaches for managing resources for the escalation of new requirements using historical data on escalations.

The fifth and sixth paper, similar to the first and second papers, dealt mainly with product and process risks, helping to proactively identify changes in existing requirements and potential injection of new ones. This decreased uncertainty of planning and staffing processes. Such information, if obtained early in the development cycle, can help decrease project risk by highlighting potential resource constraints.

Software Engineering remains a relatively new field with broad scope for quantitative methodological work as well as the development of quantitative concepts. This thesis plays a role in injecting some of these elements into the field, but many open questions remain to which methods such as those developed here will prove invaluable.

For example, identifying problematic requirements early in the development cycle, improving automatic identification of rediscovered defects and, finally, determining early signs of product quality deterioration and deriving actions needed to restore the quality. We are looking forward to making continued contributions to these challenging and societally important problems.

References

[1] E. M. Hall, *Managing Risk: Methods for Software Systems Development*. Addison-Wesley Professional, 1998.

Curriculum Vitae

Name:	Andriy Miranskyy
Post-secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada 2004-2011 Ph.D.
	The University of Western Ontario London, Ontario, Canada 2002-2004 M.Sc.
	National Technical University Kiev, Ukraine 1998-2001 Specialist
	National University Kiev, Ukraine 1995-1998 B.Sc.
Honours and Awards:	IBM Tech Connect 2009: Judge's Pick Award 2009
	MITACS Student Awards Program for Outstanding Service to the Student Network 2009
	CasCon Best Student Paper Award 2008
	IBM First Patent Application Invention Achievement Award 2007
	IBM Ph.D. Fellowship Award 2006-2008
	University of Western Ontario Graduate Teaching & Research Assistantship International Graduate Student Scholarship 2002-2007

Related Work Experience	DB2 QA Developer and Analyst IBM Canada Ltd. 2008-2011
	Teaching Assistant The University of Western Ontario 2002-2008
	Sales Manager Givaudan SA 2000-2002
	Software Developer F. Hoffmann - La Roche Ltd. 1998-2002
Service	Communication Officer Executive Committee, Networks of Centers of Excellence Trainee Association (NCETA), http://www.nce.gc.ca 2006-2008
	Ontario Representative, NCETA liason Student Advisory Committee, Mathematics of Information Technology and Complex Systems (MITACS) Network of Centers of Excellence, http://www.mitacs.ca 2006-2008
	Councilor Society of Graduate Students, The University of Western Ontario 2005-2006
Professional Activities	Co-chair of the "*Quality, the Critical Evolution for Software Development Education, Business, User Satisfaction and Career Success in the 21st Century*" workshop held at CASCON 2006 Conference, Toronto, Canada

List of Refereed Publications

A.V. Miranskyy, M. Davison, M. Reesor, and S.S. Murtaza: *Using entropy measures for comparison of software traces*, submitted to Information Sciences

B. Caglayan, A. Tosun, A.V. Miranskyy, A. Bener, and N. Ruffolo. Understanding the Explanatory Power of a Defect Prediction Model For Different Defect Categories, *International Journal of Empirical Software Engineering*, (2011) (Accepted)

Z. Li, N.H. Madhavji, S.S. Murtaza, M. Gittens, A.V. Miranskyy, D. Godwin, and E. Cialini. Characteristics of multiple-component defects and architectural hotspots: A large system case study. *International Journal of Empirical Software Engineering*, (2011) (Accepted)

A. Tosun, B. Caglayan, A.V. Miranskyy, A. Bener, and N. Ruffolo, *Different Strokes for Different Folks: A Case Study on Software Metrics for Different Defect Categories*, Second International Workshop on Emerging Trends in Software Metrics (WeTSOM'11), (2011) (Accepted)

B. Caglayan, A. Tosun, A.V. Miranskyy, A. Bener, and N. Ruffolo. *Usage of multiple prediction models based on defect categories*. In Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE '10). Article 8, DOI=10.1145/1868328.1868341 (2010), 9 pages

Z. Li, M. Gittens, S.S. Murtaza, N.H. Madhavji, A.V. Miranskyy, D. Godwin, and E. Cialini, *Analysis of pervasive multiple-component defects in a large software system*. In Proceedings of the 25th IEEE Int'l Conference on Software Maintenance (ICSM'09), (2009), 265-273

A.V. Miranskyy, E. Cialini, and D. Godwin. *Selection of customers for operational and usage profiling*. In Proceedings of the Second International Workshop on Testing Database Systems (DBTest '09). Article 7, DOI=10.1145/1594156.1594165 (2009), 6 pages

A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding, and D. Godwin, C.A. Taylor, *SIFT: A Scalable Iterative-Unfolding Technique for Filtering Execution Traces*, In Proceedings of the 2008 Conference of the Center For Advanced Studies on Collaborative Research: Meeting of Minds (CASCON '08), DOI=10.1145/1463788.1463817 (2008), 15 pages

C. A. Taylor, M. S. Gittens, and A.V. Miranskyy, *A case study in database reliability: component types, usage profiles, and testing*. In Proceedings of the first international Workshop on Testing Database Systems (DBTest '08), Article 11, DOI=10.1145/1385269.1385283 (2008), 6 pages

A.V. Miranskyy, N.H. Madhavji, R. Ferrari, S. Ghobrial, C.D. Giaraffa, Q.A. Rahman, *Requirements Escalation*, in Proceedings of the Workshop on Measuring Requirements for Project and Product Success (MeReP), November 2007, http://www-swe.informatik.uni-heidelberg.de/home/events/MeReP.htm

M. Gittens, D. Godwin, E. Cialini, A. Miranskyy, M. Wilding, C. Taylor, *Reality-based QA, a story of software profiling success*, in Proceedings of the IBM (refereed) Conference on Software Engineering for Tomorrow (SWEFT-2007). IBM T.J Watson Centre New York, USA. October, 2007

A.V. Miranskyy, M.S. Gittens, N.H. Madhavji, C.A. Taylor, *Usage of Long Execution Sequences for Test Case Prioritization*, To appear in the Fast Abstracts of the 18th IEEE International Symposium of Software Reliability Engineering, 2007

A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding, D. Godwin, *An Iterative, Multi-Level, and Scalable Approach to Comparing Execution Traces*, in Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, (2007), 537-540

A. Miranskyy, N. Madhavji *Managing Requirements Invalidity Risk*, in Proceedings of Workshop on the Interplay of Requirements Engineering and Project Management in Software Projects (REProMan), (2005)

A. Miranskyy, N. Madhavji, M. Davison, M. Reesor, *Modelling assumptions and requirements in the context of project risk*, in Proceedings of the 13th IEEE International Conference on Requirements Engineering, (2005), 471-472

O.M. Rozhmanova, E.V. Dolgaya, L.N. Stelmakh, S.V. Vasilovskaja, I.A. Votjakova, N.A. Kudrja, A.V. Miranskyy, I.S. Magura, *Effect of interferon-α2b on the cells of human embryonic nerve tissue in neurogenesis*, Neurophysiology **36** (2004), no. 5-6, 319-324.

O.M. Rozhmanova, E.V. Dolgaya, L.N. Stelmakh, N.Kh. Pogorelaya, V.V. Kucher, A.V. Miranskyy, I.S. Magura, G.Kh. Matsuka, *Sodium transport in the human neuroblastoma cells during early phase of differentiation with recombinant interferon-α2b (laferon)*, Biopolymers and cell, **16** (2000), no. 6, 540-546, in Russian.

E.V. Dolgaya, O.M. Rozhmanova, L.N. Stelmakh, A.V. Miranskyy, Yu.I. Kudryavets *Effects of interferon-α/β on Ca^{2+} influx and binding in murine thymocites*, Biopolymers and cell, **16** (2000), no. 3, 225-228, in Russian.

A.V. Miranskyy and A.I. Shapiro, *Frozen pool*, Quant (Scientific popular physical and mathematical journal), (1995), no. 4, 46-47, in Russian.

Non-Refereed Publications

A.V. Miranskyy, E. Cialini, and D. Godwin, *Selection of Customers for Operational and Usage Profiling*, IPCOM 000203574D, (2011)

A.V. Miranskyy, M. Davison, M. Reesor, S.S. Murtaza, *Using entropy measures for comparison of software traces*, CoRR abs/1010.5537, (2010)

A.V. Miranskyy and D. Godwin, *Trend change analysis using inflection points detection*, IPCOM 000177081D, (2008)

A.V. Miranskyy, N.H. Madhavji, R. Ferrari, S. Ghobrial, C. D. Giaraffa, Q.A. Rahman, Managing *Requirements Escalation*, TR-74.210, IBM Center for Advanced Studies (CAS), Toronto, (2007). Also available as Technical Report #706, Department of Computer Science, University of Western Ontario, Canada.

A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding, and D. Godwin, *An Iterative, Multi-Level, and Scalable Approach to Comparing Execution Traces*, TR-74.209, IBM Center for Advanced Studies (CAS), Toronto, (2007), https://www.ibm.com/ibm/cas/publications/index.shtml. Also available as Technical Report #686, Department of Computer Science, University of Western Ontario, Canada.

A. Miranskyy, N. Madhavji, M. Davison, M. Reesor, *Modelling assumptions and requirements in the context of project risk*, Technical Report #645, Department of Computer Science, University of Western Ontario, Canada (2005).

A. Miranskyy, *Pricing Defaultable Bonds and Options in a CIR Risk & Default Framework*, M.Sc. thesis, University of Western Ontario (2004).

Posters And Demos

A.V. Miranskyy, M. Gittens, N. Madhavji, C. Taylor, M. Wilding, D. Godwin, *Usage of Long Execution Sequences for Test Case Prioritization*, CASCON 2007, Toronto, ON, Canada, (2007)

A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding, and D. Godwin, *An Iterative, Multi-Level, and Scalable Approach to Comparing Execution Traces*, the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2007

A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding, and D. Godwin, *An Iterative, Multi-Level, and Scalable Approach to Comparing Execution Traces,* MITACS 2007, Winnipeg, MB, Canada, (2007)

A.V. Miranskyy, N.H. Madhavji, M.S. Gittens, M. Davison, M. Wilding, and D. Godwin, *Test coverage analysis,* CASCON 2006, Toronto, ON, Canada, (2006)

A. Miranskyy, N. Madhavji, M. Davison, M. Reesor, *Modelling assumptions and requirements in the context of project risk,* 13th IEEE International Conference on Requirements Engineering, Paris, France, (2005)

A. Miranskyy, N. Madhavji, M. Davison, M. Reesor, *Modelling assumptions and requirements in the context of project risk,* MITACS 2005, Calgary, AB, Canada, (2005)

Patents

A.V. Miranskyy, D. Godwin, E.Cialini, A technique for estimation of confidence interval for probability of defect rediscovery, United States Patent & Trademark Office's Application, (2009)

M. Davison, M.S. Gittens, D. Godwin, N.H. Madhavji, A. Miranskyy, and M. Wilding, "*Computer Software Test Coverage Analysis*", United States Patent & Trademark Office's Patent # 7793267, (2006)

Presentations

"Selection of customers for operational and usage profiling", DBTest'09, Rhode Island, USA, (2009)

"Profiling, from the Bottom Up", Workshop on Software Success: a Sum of Customer Details, CASCON 2007, Toronto, Canada, (2007)

"A User-Centered Approach to Improving System Testing", University of Western Ontario Research in Computer Science Conference, UWORCS, (2006)

"Managing the Requirements Lifecycle", Workshop on Software Requirements for Large-Scale Development Projects, CASCON 2005, Toronto, Canada

"The Reality of Defect Prediction Models", Workshop on Quality-Based Process and Cost-Effective Project Management, CASCON 2005, Toronto, Canada

"Efficient algorithm for computing quantiles of the noncentral chi-squared distribution", MITACS NCE Risk & Finance - Theme Meeting, University of Calgary, Calgary, AB, (2005)