Network Analysis of Large Scale Object Oriented Software Systems

Doctor of Philosophy

Anjan Pakhira

26-February 2013

Newcastle University

SCHOOL OF COMPUTING SCIENCE

Abstract

The evolution of software engineering knowledge, technology, tools, and practices has seen progressive adoption of new design paradigms. Currently, the predominant design paradigm is object oriented design. Despite the advocated and demonstrated benefits of object oriented design, there are known limitations of static software analysis techniques for object oriented systems, and there are many current and legacy object oriented software systems that are difficult to maintain using the existing reverse engineering techniques and tools. Consequently, there is renewed interest in dynamic analysis of object oriented systems, and the emergence of large and highly interconnected systems has fuelled research into the development of new scalable techniques and tools to aid program comprehension and software testing.

In dynamic analysis, a key research problem is efficient interpretation and analysis of large volumes of precise program execution data to facilitate efficient handling of software engineering tasks. Some of the techniques, employed to improve the efficiency of analysis, are inspired by empirical approaches developed in other fields of science and engineering that face comparable data analysis challenges.

This research is focused on application of empirical network analysis measures to dynamic analysis data of object oriented software. The premise of this research is that the methods that contribute significantly to the object collaboration network's structural integrity are also important for delivery of the software system’s function. This thesis makes two key contributions. First, a definition is proposed for the concept of the *functional importance* of methods of object oriented software. Second, the thesis proposes and validates a conceptual link between object collaboration networks and the properties of a network model with power law connectivity distribution. Results from empirical software engineering experiments on JHotdraw and Google Chrome are presented. The results indicate that five considered standard centrality based network measures can be used to predict functionally important methods with a significant level of accuracy. The search for functional importance of software elements is an essential starting point for program comprehension and software testing activities. The proposed definition and application of network analysis has the potential to improve the efficiency of post release phase software engineering activities by facilitating rapid identification of potentially functionally important methods in object oriented software. These results, with some refinement, could be used to perform change impact prediction and a host of other potentially beneficial applications to improve software engineering techniques.

Acknowledgements

The last three years have been one of the most challenging, exciting, and stimulating periods of my academic life, and I owe a lot of thanks and gratitude to many who have contributed to my growth and celebrated in the process.

I would like to express my sincere thanks to Peter Andras who gave me the opportunity to pursue this research. Peter has been a constant source of inspiration, guidance, and motivation all through this research.

I have had the opportunity to interact with Andrian Marcus and Wahab Hamou-Lhadj on the research topic and have benefitted from their years of experience and insight.

I would like to thank my wife and toddler son who have shown patience and understanding and provided me with the support without which this work would not have been possible.

Finally, I would like to thank my father, who has been a friend, philosopher and guide, believing in me and facilitating this journey.

This work was made possible because of EPSRC funded PhD studentship, and support provided by the School of Computing Science, Newcastle University, for which I will always be grateful.

Contents

Abstract I

Acknowledgements III

Contents IV

List of Figures VIII

List of Tables IX

Chapter 1. Introduction 1

1.1 Motivation 1

1.2 Aim, objectives and contributions 4

- 1.2.1 Aim 4
- 1.2.2 Objectives 5
- 1.2.3 Contributions 6

1.3 Outline of the thesis 6

Publications 7

Chapter 2. Background 8

2.1 Introduction 8

- 2.1.1 Scheme of review and tables of classifications 9

2.2 Program analysis overview 11

- 2.2.1 Static analysis 15
  - Software inspection and defect detection 18
  - Object oriented design 20
  - Feature location 23
  - Design patterns and method stereotypes 25

2.3 Network analysis 27

- 2.3.1 Network Graph theory basic definitions and measurements 27
- 2.3.2 Complex networks 29
  - Erdős-Rényi (ER): Random graphs 29

Watts-Strogatz (WS): Small-world networks 30
Barabási-Albert (SF): Scale-free networks 32
Network analysis measures 34
2.3.3 Use of Complex network modelling 38
Social and socio-technical systems 39
Computational and systems biology 40
Complex networks and program analysis 41
2.4 Use of the Cloud 43
Chapter 3. Review of Dynamic Analysis Literature 47
3.1 Introduction 47
3.2 Dynamic analysis 47
3.2.1 Software testing and profiling 50
3.2.2 Data collection 51
3.2.3 Object oriented design quality 54
3.2.4 Program comprehension 54
3.2.5 Complex networks based modelling 57
3.3 Mixed mode analysis 58
3.3.1 Software testing and profiling 58
3.3.2 Data collection 59
3.3.3 Program comprehension 59
3.4 Empirical experiment in software engineering 61
3.5 Network analysis 63
3.5.1 Complex networks and program analysis 63
3.6 Review notes 65
Chapter 4. Concept of functional importance 67
4.1 The concept of Functional Importance 67
4.2 Definition of functional importance 70
4.3 Functional importance and functional requirements 70

4.4 Practical verification of functional importance 71

4.5 Research questions 72

Chapter 5. Experiment design and techniques 74

5.1 Introduction 74

5.2 Experiment design 76

5.2.1 Verification of scale free power law distribution 76

Verification of Power Law 77

5.2.2 Predicting functional importance 79

Network analysis measures 80

Producing a ranked list of functionally important elements 82

5.2.3 Choice of subject software 83

5.2.4 Collection and management of data 84

Manual instrumentation 85

Tool driven instrumentation 86

Source code instrumentation 86

Token based tools 86

Special compiler based tools 86

Static Binary instrumentation of executable systems 87

Dynamic Binary instrumentation of executable systems 87

Process interference, injection and process sluggishness 90

Program preparation 90

Managing data 90

5.2.5 Data analysis using Network Analysis 91

Generation of network representation 91

5.2.6 Verification of network analysis results 93

Generation of mutant programs 94

Verification of functional importance 95

Producing random lists to establish a baseline for comparison 95

5.2.7 Experiment workflow 95
5. 3 Discussion 96
Chapter 6. Experiments on JHotdraw 97
6.1. Introduction 97
6.2 Data from execution trace 100
6.3 Network Analysis of data 101
6.3.1 Vanilla analysis of network measures 101
Discussion 104
6.3.2 Combined analysis of network measures 107
6.4 Discussion 110
Chapter 7. Experiments on Google Chrome 114
7.1 Introduction 114
7.1.1 Google Chrome 114
7.2 Data from execution profile trace 115
7.3 Network Analysis of data 116
7.3.1 Use of the cloud in mutation testing work flow 118
7.4 Discussion 124
Chapter 8. Discussion and Conclusion 127
8.1 Discussion 127
8.1.1 The concept of functional importance 127
8.1.2 Methodology and Results 128
8.1.3 Implications 130
8.1.4 Limitations 132
Validation Technique 133
Data Generation 133
Network representation 134
Statistical analysis 134
8.2 Conclusion 134

8.3 Future work 136

8.3.1 Experiment cycle 136

Validation technique 136

Data analysis 137

8.3.2 Extension of empirical experiments 137

Bibliography 139

List of Figures

Figure 2.1 Schematic illustration of program analysis based on techniques employed.. 14

Figure 2.2 Illustration of data used in program analysis (binary format logs and traces not shown). 14

Figure 2.3 Schematic overview diagram of software metrics. 15

Figure 2.5 An example of a random graph (Erdős and Rényi [79, 80]). 29

Figure 2.6 Illustrative figure that shows intermediate regime between regular and random graphs where small world graphs exist [84]. 31

Figure 2.7 Graph of characteristic path length vs. clustering coefficient in small world graphs [84]. 32

Figure 2.8 Distribution of node linkages in Random (left) vs. Scale Free networks (right) [7]. 34

Figure 2.9 Illustration of graph modularity with 3 clusters, adopted from [100]. 38

Figure 2.10 Conceptual diagram of a cloud service setup on Amazon. 45

Figure 3.1 Schematic illustration of program analysis based on techniques employed.. 48

Figure 4.1. Temporal ordering graph representation of a partial usage scenario from the perspective of the user. 68

Figure 4.2 A representation of user experience (UC+, UC–) and software execution (SC+, SC–) equivalence classes. 70

Figure 5.1 Relationship between the mathematical space of statistical graph theory and the system subjected to network analysis 75

Figure 5.2 Skewed and normal distribution plots and Rank-Frequency Plot based on CDF gives exponent as α -1. 78

Figure 5.3 Trace fragments of TPTP Probekit trace and TPTP Profile trace. 91

Figure 5.4 Google Chrome trace data in SQL Lite table. 92

Figure 5.5 Google Chrome object-method calls network. 92

Figure 5.6 Diagrammatic illustration of experiment workflow. 96
Figure 6.1 Class hierarchy diagram of JHotdraw.....98
Figure 6.2 The UML diagram of core JHotdraw framework......99
Figure 6.3 The network representation of the JHotdraw message interaction network. 100
Figure 6.4 The performance of vanilla ranking produced by edge ranking methods. ..104
Figure 6.5 The performance of combination ranking from the summation of network analysis scores......109
Figure 6.6 The performance of combination ranking from the summation and product of network analysis scores......109
Figure 6.7 The performance of combination ranking from the product of network analysis scores......110
Figure 6.8 TPTP Probes using ThreadSafeTracer library......112
Figure 6.9 TPTP probe with inline use of ReentrantLock().....113
Figure 7.1 The network representation of Google Chrome. 115
Figure 7.2 Schematic diagram of the Cloud-based mutation testing. 119
Figure 7.3 The performance of the vanilla ranking of predictions of functional importance of methods of objects, produced by WCS scoring......122
Figure 7.4 The performance of the vanilla ranking of predictions of functional importance of methods of objects, produced by CFS scoring. 122
Figure 7.5 The performance of the vanilla ranking for predictions of functional importance in methods of objects produced by BWS scoring. 123
Figure 7.6 Multi Process Architecture of Google Chrome Browser [239]......125
Figure 7.7 Google Chrome Conceptual Application layers [239]. 126

List of Tables

Table 2.1 Spread of literature by year of publication.....10
Table 2.2 Literature spread by broad categories. 10
Table 2.3 Literature in program analysis spread by categories......10
Table 2.4 Literature in program analysis spread by categories......11
Table 2.5 C&K metrics suite [40] for OOD.....20
Table 2.6 Comparison of essential properties of ER,WS and SF network models [90]. 34

Table 2.7 Definition of the power-law distribution and several other common statistical distributions [91]. 35
Table 3.1 Statistical scales and valid transformation that can be applied [196]. 61
Table 3.2 Illustration of scale types, possible software measures and appropriate statistics [196]. 62
Table 6.1 Tabulation of results of verification of predictions of functional importance of JHotdraw. 103
Table 6.2 Results verification of the Top 20 ranked methods of JHotdraw. 103
Table 7.1. Top-6 predictions from network analysis of Google Chrome. 117
Table 7.2 Tabulation of the results of verification of predictions of functional importance of Google Chrome. 120
Table 8.1 Results summary of verification of functional importance predictor measures. 130

Chapter 1. Introduction

1.1 Motivation

Computers and the Internet have revolutionised life in many ways, enabling us to use, transform, and share information. At the heart of this revolution is an intangible man-made thing we call software.

In the developments from Turing's machine to Von Neumann's machine, from machine languages to higher level programming languages and many computing paradigms, the only constants have been higher levels of abstraction, more complex software systems, change, and the evolution of use and users.

Humans have written large volumes of software. These large volumes of software data in human readable and machine readable form pose a serious storage challenge, however this challenge can be addressed through the provision of more capable storage media and storage solutions.

The challenges that software engineers and computer scientists face today are not in creating innovative and complex software solutions, but in ensuring that the software is dependable, that it is designed optimally, and that the development processes are efficient and scalable. It is not sufficient to just produce a software solution but it should be constructed in a manner that facilitates efficient software testing and maintenance.

The need to measure product quality, design quality, and development process efficiency has resulted in a wide variety of metrics and measures that cover structured programming, object oriented programming, UML design, and project processes. The efficacy of these proposed metrics and measures is not clearly established in the absence of comparable empirical evidence and the lack of widely accepted standards.

Software engineering is a human participation intensive activity, making software products and services expensive to produce. Software products and services are not yet covered by robust commercial legal structures that protect consumers against poor product quality, and the consequence of product failure on the producers is typically not as severe as in the case of products from other more established fields of engineering. A result of this has been inadequate motivation for investment in the development of software testing infrastructures.

We live not just in the age of information networks, but also of software networks. For instance, consider the case of internet enabled online banking, which makes it possible for us to gain access to our bank accounts and associated services on our mobile computing devices or desktop computing devices, without having to walk into a bank. Recently a large UK based banking group was unable to provide these service as a result of a software malfunction, which left the customers without access to full banking services whether online or at the branch [1]. This resulted in financial loss due to the disruption of normal operations spanning several weeks. Such failures are not exceptional, in fact software failures are more common than is acknowledged, representing economic losses totalling 50-60 Billion US Dollars annually according to a report from the National Institute of Standards and Technology (NIST) on this subject [2].

In the last two decades the prominent emergence of the Internet, new innovations in digital storage technology, and further advances in computer processor technology has resulted in a huge growth in the volume of data being produced, processed, and consumed. Software systems have evolved not only in size but in the complexity of composition. Whilst software engineering tools, techniques, and technology have evolved, there is concern that the pace of this growth has not been sufficient to address the significant engineering challenges that these large, complex, and highly interconnected software systems present [3].

It is widely acknowledged that it is difficult to understand real world natural, physical and man-made systems characterised by increasing size and complexity. Research on the various phenomena of these systems, as a means to gain understanding and support reasoning, has progressed along various parallel approaches with varying degrees of success. One of these approaches is to view these systems in light of complex systems theory, which has been influenced by theory of general systems. The basic premise is that the dynamical behaviour of these systems is greater than the sum of its parts.

One of these emergent approaches in complex systems theory is Complex Networks modelling. This approach provides a skeletal framework to represent real world systems as a network graph of essential system objects and pair-wise interaction relationships amongst these objects. This skeletal framework can be used as the basis to analyse the dynamic and non-dynamic properties of the system being modelled, by over laying this framework with temporal information. It is also possible to analyse the evolutionary and growth properties of the system using this modelling technique. This skeletal framework also facilitates reasoning based on correlation between the systems functional integrity and the structural integrity of the underlying network graph representation.

Statistical physics [4] of these network models enable researchers to answer such questions as; what are the most important system objects, how robust to failure is the system being modelled, or what are the conditions under which the modelled system is likely to lose functional integrity. The network graph itself is a mathematical model, and the reasoning is based on empirical models that need to be developed for systems that are sought to be modelled. This approach makes it possible to reason about both the global and local properties of a system and provides an avenue to take a holistic or reductionist view as appropriate, to support reasoning [5].

The challenges posed by software complexity, size, and the recognised lack of scalability of the current software engineering techniques to measure properties of interest, to isolate design, or implementation problems quickly, is the motivation that underpins this research.

Inspired by research that applies Complex Networks modelling to real world systems to discover key elements of these systems; this research hypothesises that such an approach may be applicable to software systems too. The ability to identify the importance of a system's elements in the context of the function of a system is non-trivial, especially when such a system has many elements that collaborate to deliver functionality. The other issue is that in computing science, the concept of the importance of system elements may be intuitive but the precise semantics of what it means to be important and the properties that enable the easy identification of importance do not exist.

It is however possible to provide a surrogate definition that relies on the structural properties of network graphs and user perception of correct functional behaviour of object oriented software at runtime. Such a definition can then be reasoned in the context of established computer science theory, and empirically verified using established network analysis techniques.

The rationale behind attempting to define the concept of functional importance (discussed in Chapter 4) of elements of an object oriented software system, in terms of its user observable behaviour, is that this ability to identify important elements can be used to significantly improve the efficiency of activities like program comprehension and software test prioritisation by using user acceptance testing.

While the ranking and sorting of class methods according to the importance of their contribution to the delivery of software functionality (especially the user-observable aspects of this functionality) is pragmatically used by software engineers [6], the concept of the functional importance of class methods in the context of object oriented software has so far not been systematically investigated. In contrast, the concept of functional importance [7, 8] is well established in the context of the analysis of complex biological and social systems. For example, well-defined and relatively small areas of the cortex in the brain are functionally important for the generation and understanding of human speech (for instance the Broca and the Wernicke areas of the cortex [9]). Similarly, a relatively small group of actors may play a functionally important role in maintaining successful teams of actors that can deliver highly successful movies (e.g. see the analysis of the network of movie actors [7]). Network analysis is often used to find functionally important components and other important characteristics of large-scale networks representing complex biological, social, or technological systems [7]. For example, in the case of protein interaction networks characterizing bacterial cells such analysis can identify around 80% of the critically essential proteins that are required for the survival of the organism in normal growth medium [8]. In this context, a component of the system being 'functionally important' means that this component of the system is critical for the delivery of the normal overall functionality of the system. For example, in the case of bacteria, network analysis is used to determine proteins that are functionally important for the survival of the bacterium (for instance if any of these proteins get blocked or are not produced within the cell, the bacterium dies or is unable to reproduce) [8]. In this case, the network of protein interactions is analysed. Functional importance of a protein can be predicted on the basis of the structural importance of the node in the network that corresponds to the protein. The structural importance of nodes is evaluated using a range of network analysis methods [4].

1.2 Aim, objectives and contributions

1.2.1 Aim

The primary aim of this thesis is to empirically validate discovery of functionally important methods in object oriented software. This process will also validate the notion that empirical models based network analysis measures that quantify centralities of a network's structural elements may be used to discover functionally important methods of an object oriented system.

1.2.2 Objectives

The objectives of this thesis are:

- To define the intuitive yet abstract concept of functional importance, in the context of our understanding of intra-systemic object interactions/collaborations in object oriented software systems.
- To use network analysis to analyse the network graph representation of intra-systemic object interactions/collaborations, based on dynamic execution trace of the software to predict structurally important elements of the network.
- To empirically verify that the predicted structurally important elements are truly important by exploiting the notion that structural integrity and functional integrity are correlated, i.e. structurally important elements are also functionally important elements of the subject system.

Network analysis is currently used in social, socio-technical, technological, and biological networks analysis. In many of these fields, the identification of elements of importance is a necessary part of analysis and reasoning about the dynamic phenomena of these systems. In many of these fields, it is difficult to gain access to the data and to carry out even non-destructive testing to verify predicted importance. This is especially true in animal brain networks analysis, which tries to understand the brain process link of the axonal structure of the brain to its normal and abnormal functional characteristics. Whilst current models of brain networks and assumptions about importance of structural-functional elements have been found to correlate with clinical observations, it is not possible to claim so, because the reliability of network measures is not known and verification cannot be attempted in clinical or laboratory environments because it is just not practical.

In software engineering however, data collection is relatively simpler and quicker, making it possible to not only use network analysis to predict the importance of elements, but also simulate the notion of network perturbation by leveraging the notion of code mutation. The research presented in this thesis has no direct connection with the application of network analysis for research on animal brain disease.

In my view this work has developed a generic prototype technique, which can be used to validate the discovery of functionally important methods in Object Oriented software by leveraging the concepts of mutation testing to simulate network perturbation. In the event that Object Oriented software and dynamic analysis data are considered as surrogates to animal brain networks data, the prototype empirical validation technique developed may find application in animal brain networks research.

1.2.3 Contributions

This thesis makes three contributions:

- Provide the first definition of functional importance in the context of software engineering. In fact, despite its wide use, the concept has not been defined even for Biological systems where it is widely used.
- Empirically verify that, in addition to other techniques, network analysis may be used to discover functionally important elements in software systems.
- Development of a generic prototype technique that makes it possible to empirically validate network analysis by using dynamic analysis data from Object Oriented software. This technique simulates the notion of network perturbation by leveraging the concepts of software mutation testing.

1.3 Outline of the thesis

The thesis is organised into eight chapters, including this chapter. Chapter 2 introduces the background material on program analysis, covering static analysis of object oriented software systems, software inspection, program comprehension techniques, network analysis, and cloud computing. Chapter 3 discusses the dynamic analysis of object oriented software and the use of dynamic analysis for program comprehension and software testing. This chapter also discusses the application of network analysis for dynamic analysis of object oriented software systems. Chapter 4 discusses the concept and definition of functional importance and the research questions. Chapter 5 discusses the techniques used in the experiments, covering data collection, data analysis, and experiment workflow. Chapters 6 and 7 discuss the results of the empirical software engineering experiments conducted on JHotDraw and Google Chrome software for this research. Chapter 8 is the concluding chapter, in which results and potential future research directions are discussed.

Publications

This research has produced one workshop publication and two technical reports [10-12] and these deal with the discussion presented in Chapters 3, 5, and 6. The main focus of these works deals with the validation of network analysis measures and initial development and validation of the concept of functional importance. The concept of functional importance and validation of the concept led to one workshop publication [13] and a book chapter on the application of cloud computing for software testing [14]. Chapters 2, 3, 5, and 7 are based on [13, 14]. Building on the run-time data collection and network analysis aspects of this research, a further piece of collaborative research was undertaken to explore mixed mode software analysis for characterising mismatches between intended object oriented design and its realisation. This work used network motif detection techniques and method stereotype analysis, leading to the final publication [15]. The main contribution of this thesis to [15] is in the data collection and in the network representation generation aspects discussed in Chapter 5.

Chapter 2. Background

2.1 Introduction

Software engineering challenges are highlighted by Goth [3], using large and ultra-large software systems as examples. In [3], the primary concern is the need to develop new software engineering techniques that can tackle the issues of size and complexity in a scalable and cost effective manner to ensure systems are reliable. The discussion follows two main strands: software development process efficiency and the engineering of large and complex software. In response to these challenges, this thesis proposes the concept of the functional importance of methods of object oriented software, and a technique to discover functionally important methods by using offline analysis of large and complex software execution (dynamic analysis) data through network graph modelling. The motivation behind using this technique is the successful application of such network modelling to comparable large and complex data in other fields of science.

The proposed concept and techniques can help software engineers reduce the time spent on manually searching complex and large code, and to prioritize testing efforts. The idea is that if the search effort is minimised, then the inefficiency of the process is likely to be reduced and time spent on testing and software maintenance can also be reduced.

Improvement of process efficiency is a possible outcome, but it is not an immediate concern of this research. However, process efficiency is a motivator, and (therefore) it is important to briefly review and to explore the various software development life cycle process issues, and the issues related to the size and complexity of software that are common to a wide range of research directions closely aligned to this work. These topics may be considered as common background for most program analysis research.

The strongest motivator of this work is the empirical validation of the concept of functional importance, and of the core assumption that network analysis measures can be used to predict functionally important methods in large object oriented software. The proposed concept of functional importance for object oriented systems, and the network analysis based technique to identify functionally important classes and methods based on dynamic analysis, are both novel ideas that need to be validated on software before the benefits of using this software analysis technique can be applied to improve the efficiency of software engineering process cycle activities like program comprehension and software test prioritization

2.1.1 Scheme of review and tables of classifications

The topics explored in this review are primarily program (dynamic) analysis in the context of large scale (object oriented) software systems, complex network analysis applied to program analysis, and cross-disciplinary influences. This review does not cover formal methods, embedded systems, or real time software systems, though references are made where these are relevant.

The review covers 322 published research papers including book chapters. All the topics covered have relevance to program analysis and span a wide timescale of almost 80 years. The numbers of papers that belong to a 40-year period (1930-1970) are only seven, and the following 20-year period (up to 1990) contains 38 papers. The period of approximately 20 years starting from 1990 contains 277 papers (see Table 2.1). The papers span eight broad topics, whose classifications can be refined (see Table 2.2). The papers are further classified based on whether they discuss static analysis, dynamic analysis, mixed mode analysis, and a fourth category for those that cannot be classed in any of the preceding (see Table 2.3).

In Table 2.4, the papers reviewed have been classified based on the technique discussed. Four categories have been used: papers that discuss techniques based on; 1) classical graph theory, 2) complex networks, 3) other non-graph theoretic techniques that cannot be classified into the first two categories, and 4) techniques are partly based on graph theory.

The classification of papers by year in Table 2.1 uses three categories, early research, recent research, and current research. There may be differing views on this categorization. In this review, recent research starts from 1970. The rationale behind using this 20-year period is that work in this period continues to have conceptual relevance and is mentioned in several current references on program analysis. The period from 1990 to the present has been classified as Current Research because the first discussions on object oriented design appear to originate around 1987-88, followed by a large number of publications from 1990 onwards.

By year:

Year of publication	**Category 1**	**Number of references**
1930 – 1970 (40 years)	ER (Early research)	7
1970 – 1990 (20 years)	RR (Recent research)	38
1990 – 2010 (20 years)	CR (Current research)	277

Table 2.1 Spread of literature by year of publication.

By broad subject/topic:

Subject/Topic	**Category 2**	**Number of references**
Computer Architecture	CA	4
Computer Science	CS	20
Software Engineering	SE	238
Psychology in Computer Science	CP	4
Information science/systems	IS	2
Mathematical/statistical in Software engineering	SM	37
Computational Biology	CB	16
General theory influencing CS	GT	1

Table 2.2 Literature spread by broad categories.

By specific topic in program analysis:

Specific topic in program analysis	**Category 3**	**Number of references**
Static analysis	SA	82
Dynamic analysis	DA	51
Mixed mode	MA	120
Not applicable	NA	69

Table 2.3 Literature in program analysis spread by categories.

By technique used/discussed:

Technique used	**Category 4**	**Number of references**
Classical Graph Theory	GT	18
Complex Network	CN	72
Not Graph Theory/Not Complex Networks	OT	214
Mixed with Classical Graph Theory	OT/GT	18

Table 2.4 Literature in program analysis spread by categories.

This research deals with program analysis of object oriented systems. One of the aims of this review is to understand whether there is a gap in analysis techniques that may be addressed by the use of complex networks modelling of software program data. The following sections of the review presents the background research on program analysis using static analysis techniques, focusing primarily on program comprehension, software testing, object oriented metrics, and complex network analysis.

2.2 Program analysis overview

The evolution of software, from small scale to large scale programs, has made the comprehension of programs, and consequently the maintenance of programs, challenging, and made the task of constructing dependable systems complex. The problems associated with efficiently performing software testing, program comprehension, and software maintenance at the product level is often linked to the software development process.

Adherence to best practice in software development processes is expected to make the development of software manageable, and it is also expected to result in the production of associated documentation that aids program comprehension and software maintenance related activities. It is now widely recognised that in practice, development of large scale software typically follows an iterative pattern, similar to the Spiral model proposed by Boehm [16], and other models such as the Waterfall model proposed by Royce [17] and the V model proposed by Forsberg and Mooz [18]. An associated concept is that of software evolution discussed by Lehman [19] through the SPE model of software programs. The central idea is that S, P, and E represent three types of software with a characteristic evolution pattern based on complexity of program design. As a software program evolves through an iterative process, design, source code, and associated documentation should theoretically be kept up to date to reflect any relevant changes. In practice this does not typically happen [20], which impedes the rapid identification of the source of problems when they occur, and necessitates the analysis of program code, documentation, and information from execution.

Iteration in the software development process introduces a nonlinear relationship between the time taken for the software development process to complete and the cost of the resultant product[16]. The primary reason behind the iterative nature of the software development process is the need to converge requirements, specification, design, and the software produced, and this includes deviations introduced by the discovery of bugs or other defects in the software [19, 21].

The length of time that a software development project may require is often determined by various factors, such as requirements, design and perceived complexity of the system, resources associated with the project, and other factors such as experience of engineers and communication with stakeholders, amongst many others. It is nontrivial to predict the cost of software products or how much software maintenance tasks are likely to cost. Rules of thumb for estimation exist, and in some cases as much as 70% of the cost of software is associated with post construction software maintenance activities. Some effort and cost estimation models like COCOMO [16] exist. However, the efficacy of such models is not clearly established, specifically when these models rely on the lines of code measure [22-24] that can vary significantly, based on the technique used to count lines of source code.

Program analysis encompasses a wide range of software engineering techniques that enable software engineering challenges to be overcome, by supporting engineering decision making in an effective and cost efficient manner. This thesis predominantly deals with the dynamic analysis of object oriented software programs and application of complex networks based modelling in this context. The software engineering challenges that this research addresses are data analysis of a large volume of dynamic analysis program data. The potential applications of the research outcome are in the program comprehension process and software testing from the point of view of prioritization.

The discussion above identifies how the conceptual interplay of software process models, the notion of iteration, and software evolution and complexity makes the estimation of the effort and cost of software development projects non trivial. Program analysis enables us to capture information about a software program or product to facilitate software engineering activities, and also to make the software development process efficient through prioritization. Software testing and program comprehension are typically software development process activities that take place after the software has been constructed, for instance at the pre-release or at the post-release software maintenance stages respectively. In modern process models, where continuous integration and test driven development techniques are followed, it is common to find unit tests [25, 26] being developed simultaneously with development of the main body of program code. However, unit testing does not replace activities that deal with final post-construction software testing activities, often referred to as integration testing and user acceptance testing. The advantage of unit testing is that units of the program are tested in isolation every time a change occurs, thus reducing the time required to perform integration or user acceptance testing at the final pre-release stage. The current philosophy that guides software testing according to Gelperin and Hetzel [27] is preventative testing, although Jackson [28] points out that it is not the volume of testing but the quality of testing guided by real usage context that can improve the dependability of software.

Program analysis is an established and active field of computer science research, and topics like software testing and program comprehension are not new. It is possible to classify research and techniques in program analysis of object oriented software into two broad categories: static analysis and dynamic analysis. In practice there exists a crossover category where techniques straddle both static and dynamic analysis. Figure 2.1 presents the schematic diagram that illustrates how the various techniques fit under the broader umbrella of program analysis. The other important idea is program data, which has been referred to many times in this literature review; Figure 2.2 is a schematic illustration of the data typically associated with program analysis.

Figure 2.3 presents the schematic overview of metrics [29] pertaining to software projects and software design and code. In the context of static and dynamic program analysis, this review is primarily concerned with the analysis performed by the use of computer assisted software engineering (CASE) tools. In this regard, it must be mentioned that aside from static and dynamic analysis, it is also possible to characterise the analysis as online or offline. In the online mode, the workflow, from program data collection through data analysis and result presentation, happens in one continuous unbroken step. Such analysis requires tools that are fully integrated with all data sources and analysis engines. In this literature review such a tool was not mentioned, although their existence in industrial environments cannot be ruled out.

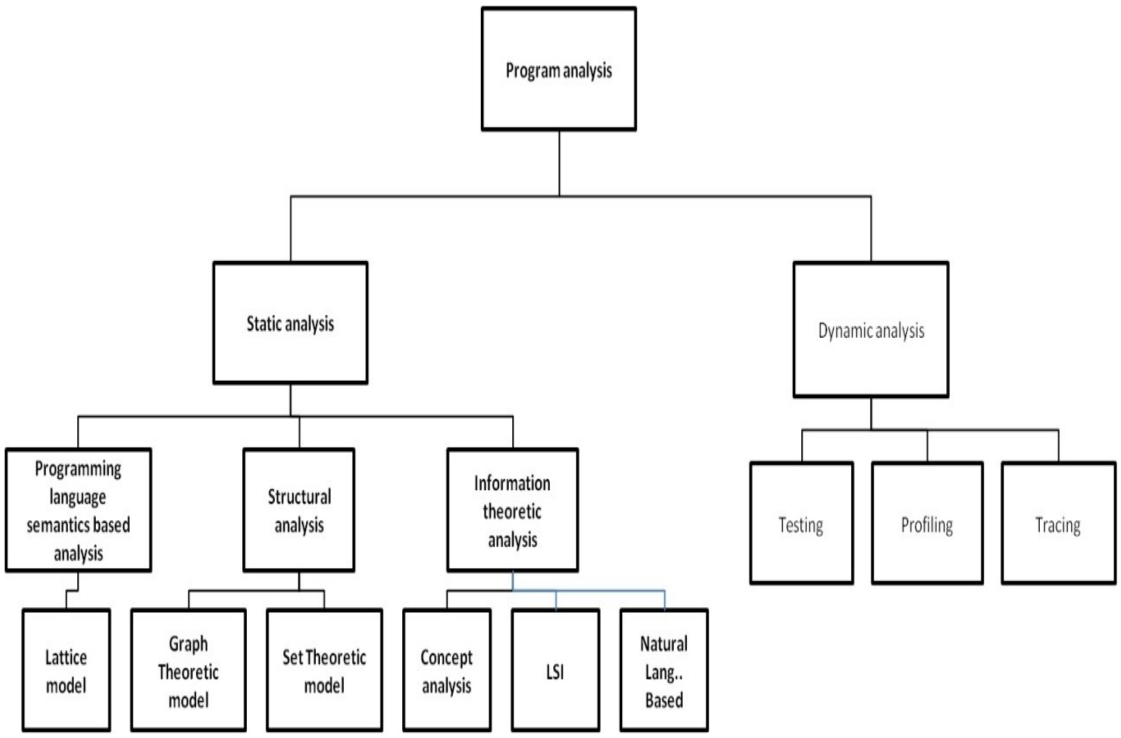


Figure 2.1 Schematic illustration of program analysis based on techniques employed

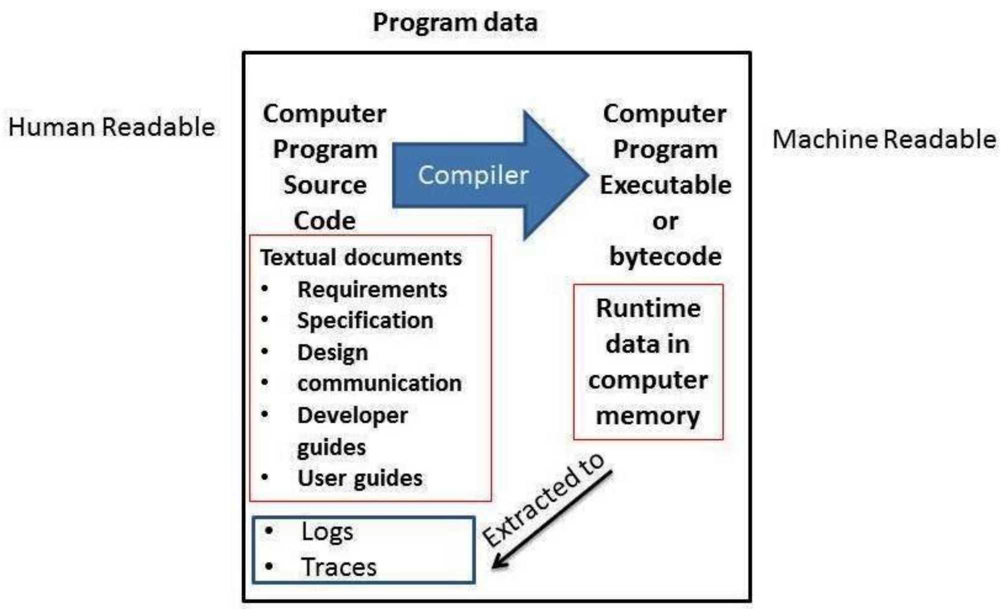


Figure 2.2 Illustration of data used in program analysis (binary format logs and traces not shown).

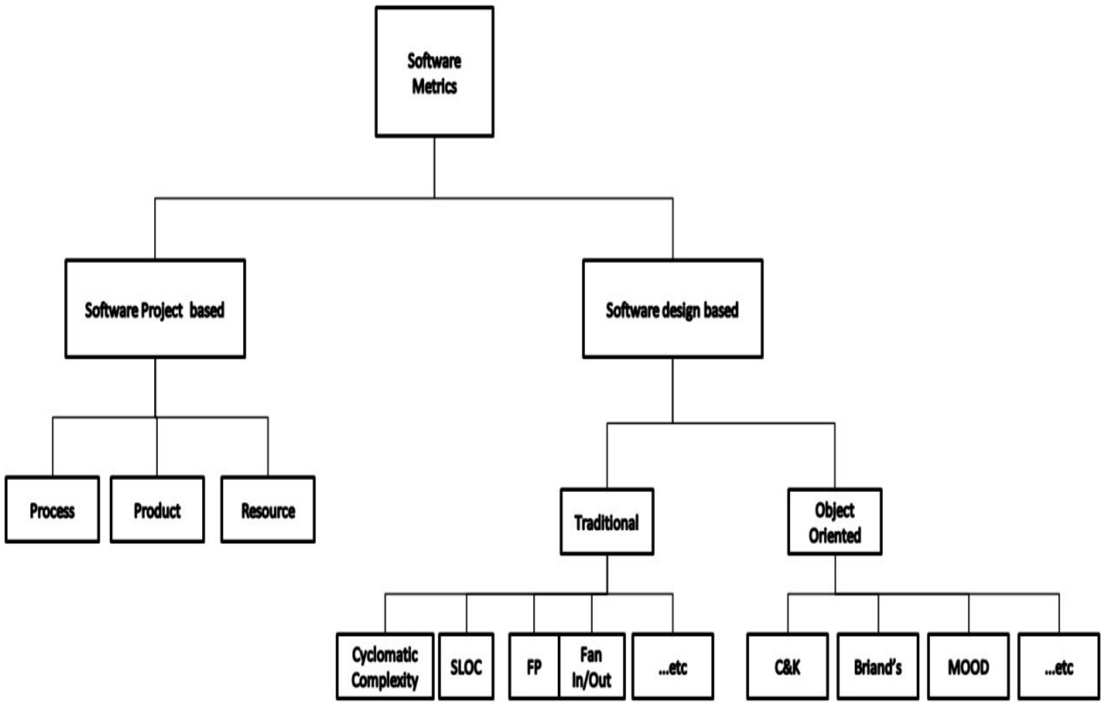


Figure 2.3 Schematic overview diagram of software metrics.

Most software analysis and measurement tools and techniques that have appeared in the literature reviewed explicitly mentioned or implied the use of offline analysis. In such analysis, the workflow containing data collection, data exploration, data analysis and result presentation is usually not implemented as one unbroken chain in the CASE tool.

Program analysis of object oriented software has primarily focused on the analysis of program structures using static analysis, and adopted most of the techniques from static analysis techniques [30-34] developed for procedural programs [35]. Despite known limitations, static analysis of object oriented systems has continued to remain popular in the absence of reliable tools to support dynamic analysis.

2.2.1 Static analysis

The fundamental notion of static analysis is program analysis that does not involve executing the program. Based on the literature review, the dominant theme in static analysis seems to revolve around modelling program data based on the semantics and syntactic rules of the programming language of construction, or to model the data such that some objects of interest may be characterised based on the global and local properties imparted by their pairwise collaboration. The models are mathematical and may be based on set theory, graph theory, statistics, or similar.

Ernst [36] defines static analysis as that which " operates by building a model of the state of the program, then determining how the program reacts to this state. Because there are many possible executions, the analysis must keep track of multiple different possible states." Therefore, it is possible to define static analysis as an approximate conservative modelling of the state space of a computer program, that may be used to analyse dynamic properties (i.e. behaviour), structural properties such as coupling and cohesion, as well as potential sources of program errors.

The other aspect of static analysis deals with utilising data mining techniques to obtain program information (available in the form of source code, and other textual documentation containing both technical and nontechnical information about a program), to support program comprehension [37], traceability link recovery [38], specification mining [39], among other similar activities.

The issue of the structural property of a program, based on the analysis of source code, suggests that at some point the notion of good programming practice began to be discussed. The concept of structured programming using higher level procedural programming languages was first formally discussed by Yourdon and Constantine in *Structured Design* [32]. This work aimed to motivate software engineers and computer scientists to follow good programming practices, write code in a modular fashion, and also discussed how one may measure three important concepts: coupling, cohesion, and code complexity.

Coupling is a conceptual property of the system that is a measure of the strength of interconnection between system modules. Therefore, coupling is an inter-modular measure.

Cohesion is a measure of intra-modular functional relatedness. Thus, a module is better designed if all the functional elements in the module are present, because they belong together and define the module’s function.

Qualitatively, modular design means the system has low coupling and each module is composed of highly functionally cohesive elements. This essentially leads to the notion of low structural complexity. That is, each module, if needed, can be independently modified without affecting other parts of the system. Such an approach to system design helps program maintenance, program comprehension, and defect isolation as well as assessment of the impact of design or module change.

In the context of object oriented software (which is the focus of this thesis) many structural design quality measurement frameworks have been proposed, of which the most prominent is known as the Chidamber and Kemerer metrics suite, or C&K metrics [40]. Many subsequent research works have fashioned their metrics on the C&K metrics model, and proposed extensions; influential among these are the frameworks for coupling and cohesion metrics proposed by Briand [41, 42].

Object oriented metrics often depend on legacy complexity measures, two of these are Cyclomatic complexity [30] proposed by McCabe, and the FanIn-FanOut measure proposed by Henry and Kafura [33].

McCabe's paper [30] demonstrates, through a series of empirical experiments on FORTRAN programs, that program complexity is not a function of size but of the decision structure of the program alone. Henry and Kafura [33] proposed structural measures of procedural large scale software systems, based on information flow analysis. Cyclomatic complexity [30] and FanIn-FanOut measures [43] are both based on the use of a control flow graph extracted from program source code, as opposed to program execution data (used in this thesis). A recurring concept that finds mention in static and dynamic analysis literature is complexity. In discussing complexity, Kearney et al. [34] explain "complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying the software. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task." Therefore, one can view program complexity in two ways: as the interaction between software components of the computer system, and as the human interaction with computer programs.

In this context, Kearney [34] discusses the underlying conceptual relationship between McCabe [30] and Henry and Kafura's measures [33], and Halstead's [43] quality measurement approach that is based on static lexemic analysis of the vocabularies of operators and operands, and the number of occurrences of each operators and operands, in the source code of computer programs.

In the context of the discussion on Halstead's work [43], Misek-Falkof's work [44] on unifying the software science approach and the computer assisted text analysis approach based on lexemic analysis is relevant, because it introduces the basis of how associated program data may be parsed to facilitate the use of natural language based querying to support data mining.

In contrast to the dynamic analysis techniques used in the empirical work in this thesis, static analysis techniques are predominantly based on text analysis techniques that are conceptually similar to the approach adopted by Halstead's work [43] and Misek-Falkof's work [44]. Static analysis is a broad area of research. In the following sections, the review focuses on static analysis approaches for program comprehension and software testing that have informed and influenced the research presented in this thesis.

Software inspection and defect detection

Manual or tool driven software inspection is a common software engineering activity. However, such activities are often ad hoc. The process oriented Fagan code inspection scheme [45, 46] in which code quality is improved by using a manual code review and inspection by teams of developers exists as an early example of a methodical approach to manual code inspection. Porter et al. [47] discuss various code inspection schemes, and try to present findings of a controlled experiment conducted in industrial settings that tries to study the effectiveness of these schemes. Porter concludes that for almost all the considered schemes of organizing source code inspection, despite the increasing number of reviewers and the varying frequency of inspections, the defect density before and after the process showed no significant improvement. Also, the rate of detection of defective code identification did not improve significantly. In large scale software development environments, manual schemes have generally been found to be ineffective because of time and resource expenses, resulting in the use of automated static analysis based techniques.

Static analysis based tools are often used to detect bad programming practices such as cloning, discussed by Kamiya in the context of the CCFind tool [48], and unused or uninitialized variables, discussed by Ogasawara in [49]. In this paper, Ogasawara describes implementation of a static analysis based software quality assurance process (called High Quality software creation) adopted at the Toshiba Corporation, and [49] reports on the effect of static analysis based software quality assurance processes on software quality improvement. Nathaniel et al [50] discuss the industrial experience of using common tools such as FindBugs for bug or defect detection. The problem discussed by most of the programs seems to be a high level of false positives in the detection. Heckman and Williams systematic review [51] on actionable alerts from static analysis tools reviews techniques for weeding out false positive alerts from static analysis tools, which are slowly getting embedded in IDE tools. Most of these tools rely on program analysis models built using Cousot's abstract analysis formal methods based techniques [52-55], Halstead's static lexemic analysis approaches [43], and sets of text processing rules. The improvement in form of actionable alerts is primarily implemented by user directed optimisation of the rules and application of the rules that the tool learns, based on acceptance or rejection of alerts by the users.

Emanuelsson and Nilsson [56] discuss the various models and theoretical underpinnings of static analysis tools, and present experience of the evaluative use of some of the widely used commercial tools on commercial software produced by Ericsson. The conclusion generally arrived at is that static analysis technology has matured to a level where these may be used instead of testing in some cases. The problems that the authors highlight are the propensity to generate reports containing a large number of false positives, and the lack of scalability of the tools and technique when handling large volumes of source code. However, the development of tools that facilitate user driven adaptive filtering, and intelligent refinement of results based on historical data, has been cited as a way to mitigate the problems. The conclusions appear to echo the conclusions reached by Nathaniel et al. in [50], which discusses a similar evaluation of static analysis tools conducted at Google.

Challet [57] discusses the bug propagation boundary of impact by considering the software function call graph's incoming and outgoing link degree distribution asymmetry. This work attempts to use scale free networks [7] to motivate discussion surrounding likely boundaries of bugs in large scale software, concluding that "fragility of software can be in part attributed to its very structure, which unfortunately seems to arise naturally from optimization considerations". In other words, the author attributes the fragility of software to be dependent on the use of optimisations in design of software, because optimisations alter the structural organisation of dependency graphs in software. Challet's study [57] uses dependency graphs and studies asymmetries in connectivity distributions at the level of functions. Hassan's paper [58] on dynamic fault prediction proposes metrics based on code change/project patterns that can be quantified as a synthetic heuristic measure. Based on this quantification risk analysis of change, proneness of code and prioritisation can be executed to guide software testing.

The discussion in the above papers presents the progressive evolution of code inspection schemes, from purely human driven manual processes to the adoption of static analysis techniques based automated tools, leading to the current practice in use of semi-automated tools that use rule based inspection, which are then optimised using human choice. The final point leads to Challet's work [57] which brings together Halstead's FanIn-FanOut measures and the complex networks based approach of data analysis that attempt to link potential software bugs to organisation of dependency graphs. This approach is novel because it attempts to use properties of a contemporary probabilistic graph theoretic model to predict where potential bugs may occur in the software and how such defects may propagate through a system. A crucial difference, when compared to the technique discussed in this thesis, is that the data capture uses non object oriented source code (Linux OS), and application of the Scale Free model does not include the verification of limiting cases.

Object oriented design

Number	**Metrics**	**Formula**
1	Weighted Methods Per Class**(WMC)**	$\boldsymbol{WMC = \sum_{i=1}^{n} c_i}$, where c_i is the complexity of methods.
2	Depth of Inheritance tree **(DIT)**	DIT = length; will be the maximum length from the node to the root of the tree.
3	Number of Children **(NOC)**	number of immediate subclasses subordinated to a class in the class hierarchy.
4	Coupling between object classes (**CBO)**	is a count of the number of other classes to which it is coupled.
5	Response For a Class **(RFC)**	$RFC = \lvert RS \rvert$ where$RS = \{M\}\ Uall\ i\{Ri\}\ where\ \{Ri\} = set\ of\ methods\ called\ by\ method\ i\ and$ $\{M\} = set\ of\ methods\ in\ a\ class$
6	*Lack of Cohesion **in** Methods **(LCOM)***	$LCOM = \lvert P \rvert - \lvert Q \rvert, if\ \lvert P \rvert > \lvert Q \rvert$ $= 0\ \ otherwise$ Where P and Q are sets of interaction instance variables of two interacting methods M_1 and M_2.

Table 2.5 C&K metrics suite [40] for OOD.

A key application of static analysis is in the analysis of the design quality of object oriented (OO) software. Design quality measures are used as indicators of potential problems in the software that are a result of poor design or poor implementation of design. The most popular metric suite proposed is the C&K metrics [40], consisting of a set of 6 metrics (see Table 2.5) to study the design quality of object oriented (**OO**) software. Chidamber and Kemerer's paper [40] not only proposes and defines the metric suite formally, but also discusses empirical validation in the context of two commercial software applications. Popularly known as C&K metrics, these metrics are widely used for static analysis of OOD. In [59], Briand *et al.*. discusses the state of empirical studies and research on object oriented metrics, concluding that despite significant research, the practical impact of the proposed metrics suites has been limited.

Briand *et al.* [59] attributes the lack of practical impact to the neglect in consideration of relevant topics such as the behaviour of object oriented software structures in the dynamic context, because the proposed metrics suites are based on analysis of single classes that fail to scale when applied to the analysis of real world OO software. Briand *et al.* [59] also highlights the weak connection between interpretation of the metrics in the context of external attributes like reliability, fault proneness, and reuse of software, as a reason behind the lack of practical impact and adoption of the vast majority of the static analysis based object oriented metrics proposed by various studies.

Two pieces of research by Briand [41, 42] formally define cohesion and coupling metrics using a set theoretic approach, and refine a very large number of similar OOD metrics. The papers empirically validate Briand's measures on a suite of OO software systems, and also improves C&K metrics [40]. The outcome is 12 measures of cohesion and 30 measures of coupling. In the context of empirical studies of quality models, Briand and Wust's research [60] is a comprehensive and critical survey of research on OOD metric sand, which puts into perspective the techniques employed by various researchers using a classification of 7 broad categories leading to the following observations:

- On design measures, the observation is that many of these measures are redundant, with only about 15 of the proposed measures being able to capture distinct design quality information. However these 15 measures are difficult to capture and require expensive analysis to compute, resulting in the continued popularity of the C&K metrics suite[40].

- Among the design measures used in the fault proneness models, coupling and size have been found to be good indicators, as opposed to cohesion measures and inheritance measures.
- Similarly for effort indicator models, size seems to be the only measure that has been found to be reliable; structural measures based models using coupling and inheritance did not bring substantial gains.
- On the predictive power of the fault proneness models, the observation is that models that use class level design measures in the prediction models, on average tend to demonstrate classification accuracy close to 80%, and find about 80% of the faults.

In [60], the following comment summarises the results: "Overall, the results suggest that design measurement-based models, for fault-proneness predictions of classes, may be very effective instruments for quality evaluation and control of OO systems." A general overarching observation made in [60] concerns the specificity of the empirical models to the environment, explaining the issue of the lack of reliable results when such models are used without careful consideration across projects.

In the case of OO software systems, numerous works have established a clear empirical relationship between class level couplings and some external attributes, such as fault proneness, by leveraging static analysis of source code, listed by Briand and Wust in [60] and [61]. It is widely recognised that such models often suffer from imprecision because of many factors, some of which are OOD related, such as polymorphism and late binding that cannot be accurately captured through static analysis. The other major cause behind imprecision has also been attributed to the tools involved, not discounting dead source code elements that result from bad software engineering practices [35].

Martin [62] discusses the issues of bad architectural design and program design in terms of some OOD principles, and proposes the I metric to quantify the architectural stability of OO software. Martin's discussion on the use of architectural and object oriented design patterns to build software systems is essentially the support of best practice in light of concepts of OOD. None of these are new; in fact, they result from awareness of the practical impact of bad design, and the process of organic evolution of software which Lehman identified in the laws of software evolution [19].

Research on applications of OOD metrics, like the C&K metric suite or Briand's metric suites, has seen development of bespoke synthetic metrics. For instance, the work by Shin et al. [61] explored the relationship between complexity, code churn, and developer activity metrics with vulnerabilities. Complexity and developer activity metrics were found to be actionable metrics. Actionable metrics are metrics that may be used by consumers of the metrics to initiate corrective actions in the context of software development projects for which these metrics have been calculated. Much of this work focuses on file change information available from source code repositories, in addition to information on initiators of these changes. Similarly, Zhang and Jacobsen [63] discuss an aspect of the mining technique based on fan-in fan-out graph, page rank, and random walk based algorithms. Two rankings are proposed for statically parsed and analysed program elements to help separate core elements and cross cutting concerns through computational approximation of the typically human driven manual process. The advantage is to reduce inconsistency in analysis, by reducing the human element and also by automating the process.

Briefly, whilst parallels may be drawn, the diversity of approaches and motivations behind the proposed OOD metrics suites and information on their empirical validation does not lend itself to comparison as to which one of these metrics is truly effective. Object oriented metrics research has seen significant empirical verification and, despite the apparently contradictory results, this has led to work on synthetic process metrics like [61], [63] and [64], that appear to be using more powerful data analysis techniques such as complex networks modelling to improve understanding of OOD and program source code data.

Feature location

A potential application of the research presented in this thesis is program comprehension. One key software engineering technique, which is often used in practice for program comprehension, is feature detection using static analysis. Dit et al. in [65] define features as follows: "In software systems, a feature represents a functionality that is defined by requirements and accessible to developers and users". So, a feature may be an operation implemented in software such as "Open File". Rajlich and Wilde in [66] provide the following working definition for concepts in the context of program comprehension in software: "Concepts are units of human knowledge that can be processed by the human mind (short-term memory) in one instance.". Concepts are considered domain knowledge related that can be associated with the distinct functionality of a system, for instance in an online financial transaction system "Online Card Payment" may be considered a concept.

Chen [67] in his paper on feature detection, proposes the concept of an abstract system dependence graph. An abstract system dependence graph contains both control flow and data flow graph elements of the system. Use of this graph facilitates concept location in a semi-automated way. Programmers may initiate computer assisted concept/feature location in conjunction with the use of dynamic analysis, to set a starting point of the concept/feature location search. Empirical validation of this feature location technique was carried out on a C language program.

Liu [68] discusses the modelling cohesion of a software system based on Information retrieval using the Latent Dirichlet Allocation (LDA) technique. This paper proposes a novel approach that uses information entropy in conjunction with information retrieval technique to measure class cohesion. The paper proposes a new metric, Maximal Weighted Entropy (MWE) that captures complementary but new dimensions of cohesion, in comparison to existing measures of cohesion. The paper reports that the result of empirical experiments and principal component analysis indicates that MWE in conjunction with LOC or LCOM (i.e. Lack of Cohesion, see Table 2.5 on C&K metrics) forms a good model for defect prediction.

In a similar context , Hill et al. [69] discusses a technique that automatically extracts natural language phrases from source code identifiers, and categorizes the phrases and search results in a hierarchy. This research [69] proposes a natural language information retrieval technique to perform a search based on contextual information as opposed to the technique based on use of verb direct objects. The technique proposed by Hill et al. [69] is comparable to the approach used by Halsted in [43] to measure the quality of a program by using static leximic analysis of vocabularies of operators and operands, and the frequency of occurrence of each class of information found in the program source code text. Misek-Falkof's [44] work attempts to reconcile Halstead's approach for programs with application of the same token counting based approach to English language text to comprehend natural languages. The parallel lies in what Hill's paper is categorizing, in order to support natural language queries to aid program comprehension. Hill uses Verb-Active Object pairs in a similar fashion to Noun-verb pairs used by Misek-Falkof [44]. Hill [69] goes further, by associating these pairs (referred to as signatures) with comment phrases. Hill's technique [69] is a recent refinement of the static analysis approach proposed in [43] and [44] , and Hill reports the improved performance of natural language queries for program comprehension, achieved by using both contextual phrases from methods and field signatures, yielding more positive results than using just Verb-Active Objects pair alone.

A related research is presented by McMillan et al. in [38], that discusses a technique which combines textual and structural analyses of software artefacts – two widely accepted techniques for recovering traceability links. In [38] McMillan et al. uses the JRipples tool for structural analysis, and Information Retrieval, LSI techniques, and F test for statistical validation. McMillan et al. reports that the proposed new technique resulted in only minor improvement over existing techniques and suggested the need for further investigation.

The above research on static analysis based feature detection discusses the need to associate features and control flow graphs that may result from user action. A key drawback of the static analysis of OO systems is that runtime effects of OOD, such as polymorphism and late binding, cannot be reliably captured by static analysis. This circumstance led Chen [67] to use a mix of static and dynamic analysis for empirical validation. The graph based data analysis used by Chen [67] is theoretically similar to the work presented in this thesis. However, there are a number of key differences. First, the work presented in this thesis is based entirely on dynamic analysis data and is likely to be more precise, albeit for a distinct scenario. Second, the focus of the thesis is on empirical verification of the concept of functional importance of methods, rather than on discovery of features.

Design patterns and method stereotypes

Design patterns are similar to solution templates for recurring design problems that facilitate communication of the design problem concisely, and also implementation of a standard programming solution. Therefore, design patterns have been referred to as standard solutions to common design problems that may be applied when such problems are identified in a system design. The seminal text on design pattern is attributed to the **Gang of Four**: Gamma et al. [62, 70] containing 23 patterns. This list of 23 has been augmented many times. The philosophy of design pattern-led software design has also been criticised [71, 72]. However, it is widely used in practice, and leads to the concept of Anti-pattern, and Bad Smells in software code.

Method stereotypes classify methods according to their functional role. Stereotypes proposed by Dragan in [73] have been classified into 3 categories: structural, collaborational, and creational. Structural stereotypes have two further subcategories: accessor and mutator. Another way of thinking about design pattern is that it provides a recipe for organising functional elements of code into units that collaborate to deliver a task in an object oriented system. Method stereotypes are a classification of the functional role fulfilled by the functional element. Knowing the stereotype of a method may facilitate program comprehension and software maintenance.

Dragan [73-75] presents first the taxonomy of method stereotypes. It then uses lightweight static analysis to extract stereotype information about OO software system. Stereotypes are presented as useful information for OO design recovery in reverse engineering. Method stereotypes facilitate classification of the method by its functional role, in contrast to design patterns that provide common compositional stencil to achieve a functional outcome. The notion proposed is that a method in an OO software can be stereotyped based on its interface signature. Each OO software has a unique distribution of stereotyped methods. In another corresponding research, Guéhéneuc et al. [76] discuss a constraint satisfaction problem (CSP) based approach, to identify complete and incomplete occurrences of design motifs in object oriented programs. In this approach, experimentally built numerical signatures of classes are prepared. These signatures characterize the classes based on their roles in design motifs. In [76], an explanation based approach was enhanced through the use of numerical signatures. The aim of this work is to improve computational performance and precision of the search for complete and incomplete use of design motifs. The results of this approach indicate that the approach proposed has more precision than a purely structural approach while preserving recall properties. The paper claims that this approach reduces instances of false positives, and has high performance and perfect recall. It also provides explanation for identified micro-architectures and interactions that may be useful to software maintainers.

Boussaidi and Mili [77] argue that an explicit representation of the design problem, solved by a design pattern, is key to supporting the three tasks in an integrated fashion. This work proposed a representation of design patterns consisting of triples <MP, MS, T> where MP is a model of the problem solved by the pattern, MS is a model of the solution proposed by the pattern, and T is a model transformation of an instance of the problem into an instance of the solution. Given an object oriented design model, model fragments are matched to MP, and when one is found, transformation T is applied, yielding an instance of MS. In effect, this approach is an approach of code transformation to replace a nonstandard design pattern model with a standard one, with potential use in automatic code generation from design models.

Program comprehension, through design pattern detection and method stereotype analysis, are both novel techniques that rely significantly on graph theoretic analysis. Similarly to Guéhéneuc et al. [76] and Boussaidi and Mili [77], one can argue that the ability to classify architectural components by using method stereotypes and pattern detection may be used as in [15] to facilitate the analysis of mismatches between design intention and realisation, and instances of mismatches can be used as triggers to prioritise program comprehension and software testing activities.

In the next sections, the literature review discusses network analysis and probabilistic graph theoretic models that have the premise to facilitate efficient analysis of real world systems, including large object oriented software.

2.3 Network analysis

2.3.1 Network Graph theory basic definitions and measurements

In mathematics and computer science, graph theory is the study of *graphs*, which are mathematical structures used to model pairwise relations between objects from a certain collection. A graph refers to a collection of objects represented as nodes (also called vertices) and a set of edges (also called arcs or ties) connecting the nodes. The edges represent the relationships between the node elements. An example of a simple graph of 6 nodes is represented in Figure 2.4. Traditionally, nodes are represented by a dot and edges by a line. In the figure below circles should be taken as dots.

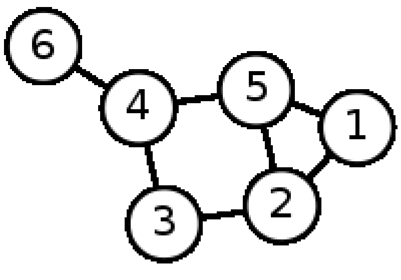


Figure 2.4 A graph containing 6 nodes.

Mathematically, a graph **G** is represented as

$G = (V, E)$ (2.1)

where V is a set of vertices and E is set of edges $E \in [V]2$ and it is assumed that $V \cap E = \emptyset$, thus elements of **E** are two-element subsets of **V** i.e. a set of vertices [78]. In Figure 2.4; $v = \{1,2,3,4,5,6\}$ is the set of vertices.$E = \{\{2,2\},\{1,5\},\{2,5\},\{5,4\},\{2,3\},\{3,4\},\{4,6\}\}$

Measures of a Graph

The number of vertices of a Graph is referred to as its order, written as **| G |**, and its number of edges is written as || G ||

Degree of a vertex is equal to the number of neighbours of a vertex v. $d_G = d(v) of\ a\ vertex\ v = number\ of\ edges\ |E(v)|\ at\ v$

Degree of a graph

$$\mathbf{d(G)} := \frac{\mathbf{1}}{|\mathbf{V}|} \sum_{v \in V} d(v) \ (2.2)$$

A path is a non-empty graph of form $= (V, E)\ where\ V = \{x_0, x_1, \dots, x_k\} and\ E = \{x_0x_1, \dots, x_{k-1}x_k\}$, where $\mathbf{x_i}$ are all distinct. The vertices $\mathbf{x_0}$ and $\mathbf{x_k}$ are linked by P and are called ends; the vertices $x_1,\dots x_{k-1}$ are the inner vertices of P. Path length is the total number of edges in P denoted by $\mathbf{P^k}$**.**

Shortest path problem refers to the problem of finding a path between two nodes in a graph, such that the sum of weights of constituent edges is minimized. In a non-weighted graph that would normally be finding the path with the least number of edges to traverse between two nodes.

The material presented above is a very brief summary of basic network graphs. Various types of simple graphs and measures are discussed in [78]. One can trace the origins of graph theory to Euler's solution of **Konigsberg's** bridge problem. Graphs have been used to model the structure of many physical, social, socio-technical, and technological systems. The models widely used belong to a family of models broadly known as regular graphs. For instance, lattices are widely used to represent molecular structure. These graphs assume a fixed number of homogeneous nodes and links, with the structure remaining static and conforming to some underlying symmetry. The problem is that large real world systems rarely contain a fixed number of homogeneous nodes that are always connected in a regular fashion. Real systems have dynamic properties that give them characteristic spacio-temporal structures. For instance, the road network of a country shrinks and expands, airline networks evolve, and social networks which are composed of humans change over time. Efforts to gain a better comprehension of these systems have led to new graph theoretic models that incorporate notions of heterogeneity, growth, and evolution. Micro and macro structural network models, functional network models, and a combination of these have been proposed in a large number of fields, prominent among them sociology, systems biology, and also computer science. Although existing measures and properties of regular graphs still apply, many of the new models have proposed empirical measures that apply only to a certain class of models, based on the statistical properties of distribution of nodes and links. These new graph theoretic models broadly constitute the field of complex networks. The data analysis in this thesis is based on the premise that complex dynamic analysis data of large OO software can be facilitated by utilising the complex network based models discussed in the following section. The basic network graph model and measures continue to remain relevant when applying complex network modelling to data analysis.

2.3.2 Complex networks

Erdős-Rényi (ER): Random graphs

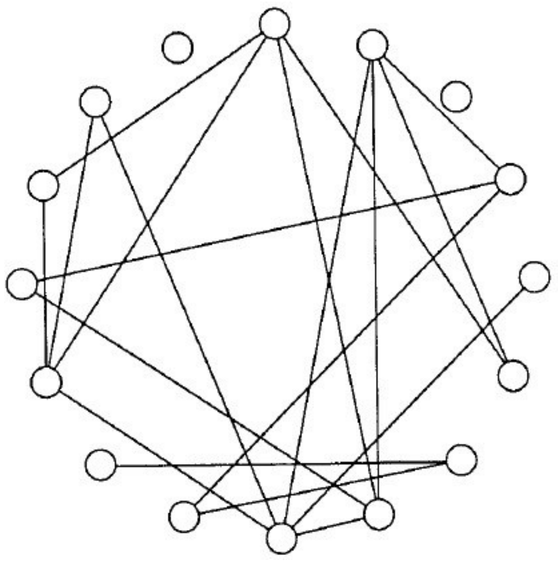


Figure 2.5 An example of a random graph (Erdős and Rényi [79, 80]).

Erdős and Rényi introduced the concept of a random graph in [79]. A random graph is simple to define. One takes a number N of nodes or "vertices" and places connections or "edges" between them, such that each pair of vertices i, j has a connecting edge with independent probability p. An illustration of a random graph, as discussed by Erdős and Rényi [79, 80], is presented in Figure 2.5. In this case, the number of vertices N is 16 and the probability p of an edge is 1/7.

This model is very popular and extensively researched in discrete mathematics. However, as a model of a real world network it has some serious deficiencies, primarily when the degree of distribution in a random graph is quite unlike what is seen in networks representative of the real world.

A vertex in a random graph is connected with equal probability p with each of the N - 1 other vertices in the graph, and hence the probability p_k that it has the degree exactly k is given by the binomial distribution:

$$p_k = \binom{N-1}{k} p^k (1-p)^{N-1-k} \qquad (2.3)$$

If average degree of a vertex in the network is considered as $z = (1 - N)p$, we can write Equation (2.3) as

$$p_k \cong \frac{z}{k!} \; e^{-z} \qquad (2.4)$$

In Eqn. (2.4) approximate equality becomes exact in the limit of large N. This distribution is the Poisson distribution. Average degree distribution of the large random graph follows the skewed Poisson distribution, which typically resembles a distorted bell shape. Bullmore and Sporns [81] and others characterise this as symmetrically centred Gaussian distribution, which is true in the case of continuous random variables. In the literature concerning complex networks, a normal approximation of Poisson distribution is often used based on application of Central Limit Theorem and the Law of Large Numbers [82].

Newman et al. [80] discuss some special cases where random graph models may be used to model some real world networks. However, in general, the research literature tends to agree that random graphs provide a poor approximation of real world networks. In this regard, Newman has analysed social (affiliation) networks, like the affiliation network of the board of directors of 1999 (Fortune 1000), and the scientific collaboration network in Physics.

Watts-Strogatz (WS): Small-world networks

Regular graphs like lattices are known to have long path lengths that increase linearly as the number of vertices increases, high clustering coefficient, and the non-random probability that determines the connection between two individual vertices. ER model random graphs have low clustering coefficient, short path length, and a constant random independent probability of connection between two vertices. Real world networks contain neither regular nor random probability of connection between vertices, where instead they have hub like topological features. That is, some vertices have more connections than other vertices. It is often found that real world networks have local triadic closures or cliques, or clusters of vertices that are connected to each other. In many real world networks, most vertices can be reached from others through a small number of edges, this is known as small world property. The most famous example in this regard has led to the idea of 6 degrees of separation, based on Milgram's famous experiment [83].

The model proposed by Watts and Strogatz in [84] is the most popular small world model characterised by its abundance of short loops. A small-world network can be constructed starting with a regular lattice of N vertices, in which each vertex is connected to k nearest neighbours in each direction, totalizing 2κ connections, where $N \gg k \gg \log(N) \gg 1$. Each edge is then randomly rewired with probability p. When $p = 0$ we have an ordered lattice with a high number of loops but large distances, and when $p \to 1$, the network becomes a random graph with short distances but few loops. Figure 2.6 illustrates the notion of the intermediate regime between regular and random graphs with increasing chance of two vertices being connected with independent random probability.

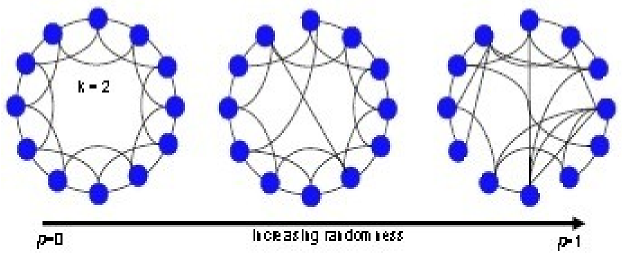


Figure 2.6 Illustrative figure that shows intermediate regime between regular and random graphs where small world graphs exist [84].

In an intermediate regime, the presence of both short distances and a large number of loops has been shown by Watts and Strogatz. The degree distribution for small-world networks is similar to that of random networks, with a peak at $\langle k \rangle = 2k$. The rise of short distances is attributed to short cut edges. That is, edges between distant vertices that help reduce the distance between two vertices that would otherwise be long.

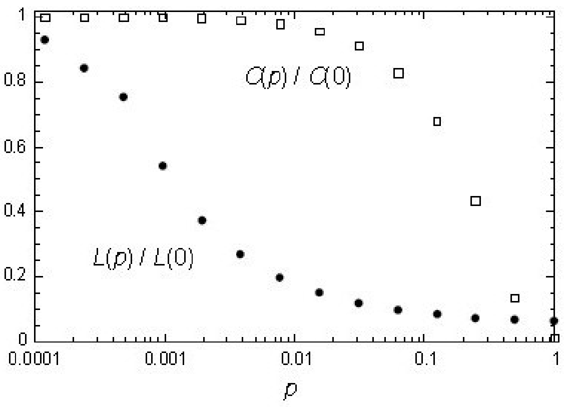


Figure 2.7 Graph of characteristic path length vs. clustering coefficient in small world graphs [84].

In Figure 2.7, it can be seen that characteristic path length $L(p)/L(0)$ and clustering coefficient $C(p)/C(0)$ show different rates of drop, with path length registering a rapid drop whilst clustering coefficient remains stable at regular graph values. This drop of path length in the intermediate region signifies the onset of small world behaviour, and the fact that clustering coefficient remains almost stable means that it is not possible to determine small world behaviour from local properties. It can also be seen that there exists a wide window of intermediate probabilities in which the small world model is true.

Research has established that network representation of a number of real world systems has properties with characteristic of WS small world networks [85]. Amaral et al. in [85] discuss the different classes of small world networks. These classifications are based on the decay connectivity in the degree distribution of small world networks. The three classes are: (a) scale-free networks, characterized by a vertex connectivity distribution that decays as a power law; (b) broad-scale networks, characterized by a connectivity distribution that has a power law regime followed by a sharp cut-off; and (c) single-scale networks, characterized by a connectivity distribution with a fast decaying tail.

Barabási-Albert (SF): Scale-free networks

The scale-free networks model was proposed by Barabási and Albert [7, 86, 87]. These networks are characterized by uneven degree distribution, in contrast to ER and WS models where vertices have random patterns of connection with a characteristic degree. In fact such a distinction may not be totally accurate, because real world networks which are characterised by growth and preferential attachment may also be in part a result of random connectivity process, and this aspect continues to be a topic of research [88]. In this model, some vertices are highly connected whilst others have few connections, with an absence of characteristic degree. The degree distribution has been found to follow power law for large k**.**

$P(k) \sim k^{-\gamma}$ (2.5)

These networks typically display the presence of hubs. That is, vertices are linked to a significant number of edges of the network.

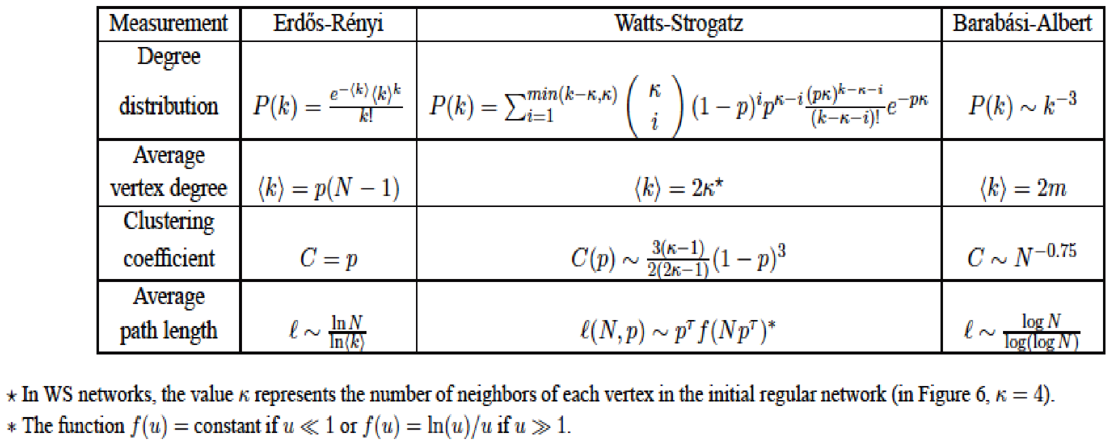
The Scale Free **(SF)** network model is based on two basic rules: growth and preferential attachment. The generation of this type of network starts with an initial set, containing m_0 number of unconnected vertices. Growth of the network happens in a step-by-step manner with the addition of new vertices. For each new vertex, m new edges are inserted between the new vertex and some previous vertex. The vertices that receive the new edges are chosen following a linear preferential attachment rule. That is, the probability of the new vertex i to connect with an existing vertex j is proportional to the degree of j

$P(i \rightarrow j) = k_j / \sum_u k_u$ (2.6) .

This is the random distribution that determines the presence of links between nodes, leading to the emergence of what may be characterised as the "rich get richer" phenomenon.

Barabási and Bonabeau in [7] provide an overview of scale free networks. The main points of the discussion can be summarised as follows: 1) These networks have some vertices that have many more links, as compared to other vertices in the networks, leading to the notion of hubs that act as nodal points in the network. 2) These networks are resilient to random attacks, especially on peripheral vertices, but are vulnerable to directed attacks on hub nodes. 3) Understanding the properties of the network structure can help us to take corrective action, so as to prevent exploitation of vulnerabilities that could lead to impairment of the system's functional integrity.

The SF network, discussed by Amaral et al. in [85], is a type of small world network. Actual classification of a network into any one of the three types is not easy, partly because of the complicated statistics involved in dealing with borderline cases. Li et al. [89] provide a critical review of the research on complex modelling that erroneously associates systems as having scale free distribution



Measurement	Erdős-Rényi	Watts-Strogatz	Barabási-Albert
Degree distribution	$P(k) = \frac{e^{-\langle k \rangle} \langle k \rangle^k}{k!}$	$P(k) = \sum_{i=1}^{min(k-\kappa,\kappa)} \begin{pmatrix} \kappa \\ i \end{pmatrix} (1-p)^i p^{\kappa-i} \frac{(p\kappa)^{k-\kappa-i}}{(k-\kappa-i)!} e^{-p\kappa}$	$P(k) \sim k^{-3}$
Average vertex degree	$\langle k \rangle = p(N-1)$	$\langle k \rangle = 2\kappa^\star$	$\langle k \rangle = 2m$
Clustering coefficient	$C = p$	$C(p) \sim \frac{3(\kappa-1)}{2(2\kappa-1)}(1-p)^3$	$C \sim N^{-0.75}$
Average path length	$\ell \sim \frac{\ln N}{\ln \langle k \rangle}$	$\ell(N,p) \sim p^\tau f(Np^\tau)^*$	$\ell \sim \frac{\log N}{\log(\log N)}$

$\star$ In WS networks, the value κ represents the number of neighbors of each vertex in the initial regular network (in Figure 6, $\kappa = 4$).
$*$ The function $f(u) = $ constant if $u \ll 1$ or $f(u) = \ln(u)/u$ if $u \gg 1$.

Table 2.6 Comparison of essential properties of ER,WS and SF network models [90].

The observed models are instances of log-normal distribution, or truncated scale free distribution, where part of the distribution may be exponential. Li et al. [89] highlights the common errors in network classification.

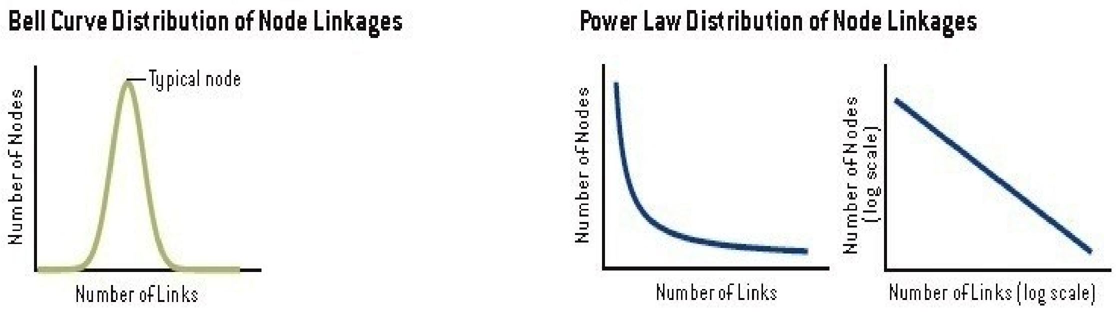


Figure 2.8 Distribution of node linkages in Random (left) vs. Scale Free networks (right) [7].

Clauset et al. in [91] discuss more reliable techniques to approach statistical and graphical classification of the network graphs generated from empirical data. Table 2.6 compares the essential properties of the three network models discussed in this section. Figure 2.8 (reproduced from Barabási and Bonabeau [7]) shows how degree distribution differs in Random and Scale Free graphs.

Network analysis measures

<u>Node degree, degree distribution, and assortativity:</u>

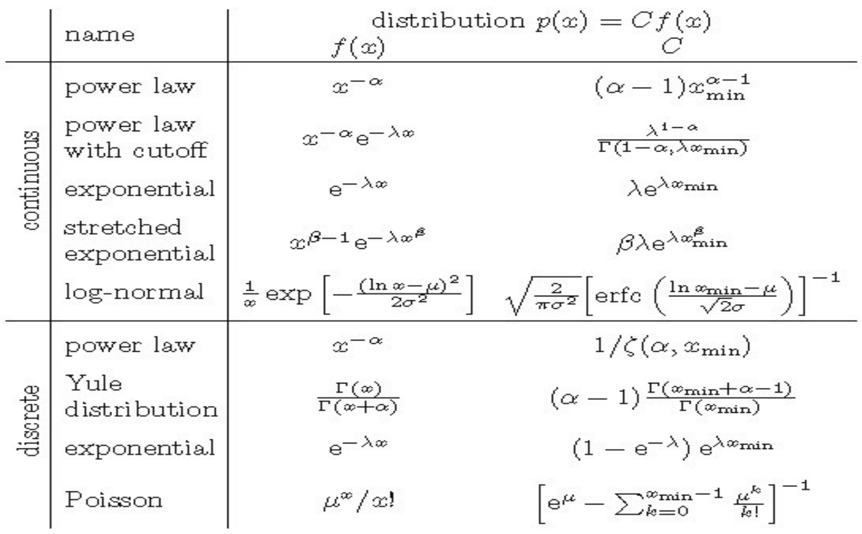
A fundamental measure of networks is node degree. Node degree (k) is the total number of connections that links a node or vertex to the rest of the network. In a directed network, one can differentiate between in-degree k_{in} and out-degree k_{out}.

$k = k_{in} + k_{out}$ (2.7)

Generally, the degree of a vertex in random networks is statistically distributed, and [92] provides the following definition and equations for degree distribution.

Considering the case of undirected networks, if vertices can be distinguished (which is normally the case in growing networks) the degree distribution of a vertex s is given by the probability p that the vertex belongs to a network of size N and has k connections (i.e. k nearest neighbours) $p(k, s, N)$. Knowing the degree distribution of individual vertices, it is possible to derive the degree distribution of the total network given by

$P(K, N) = \frac{1}{N} \sum_{s=1}^{N} p(k, s, N)$ (2.8)



	name	distribution $p(x) = Cf(x)$ $f(x)$	C
continuous	power law	$x^{-\alpha}$	$(\alpha - 1)x_{\min}^{\alpha-1}$
	power law with cutoff	$x^{-\alpha}e^{-\lambda x}$	$\frac{\lambda^{1-\alpha}}{\Gamma(1-\alpha,\lambda x_{\min})}$
	exponential	$e^{-\lambda x}$	$\lambda e^{\lambda x_{\min}}$
	stretched exponential	$x^{\beta-1}e^{-\lambda x^{\beta}}$	$\beta\lambda e^{\lambda x_{\min}^{\beta}}$
	log-normal	$\frac{1}{x}\exp\left[-\frac{(\ln x-\mu)^2}{2\sigma^2}\right]$	$\sqrt{\frac{2}{\pi\sigma^2}}\left[\operatorname{erfc}\left(\frac{\ln x_{\min}-\mu}{\sqrt{2}\sigma}\right)\right]^{-1}$
discrete	power law	$x^{-\alpha}$	$1/\zeta(\alpha, x_{\min})$
	Yule distribution	$\frac{\Gamma(x)}{\Gamma(x+\alpha)}$	$(\alpha - 1)\frac{\Gamma(x_{\min}+\alpha-1)}{\Gamma(x_{\min})}$
	exponential	$e^{-\lambda x}$	$(1 - e^{-\lambda})\, e^{\lambda x_{\min}}$
	Poisson	$\mu^{x}/x!$	$\left[e^{\mu} - \sum_{k=0}^{x_{\min}-1}\frac{\mu^k}{k!}\right]^{-1}$

Table 2.7 Definition of the power-law distribution and several other common statistical distributions [91].

Clauset et al. in [91] provide a table of power-law and other common distributions found in complex networks (see Table 2.7).

Assortativity is the correlation between the degrees of connected nodes. Positive assortativity indicates that high-degree nodes tend to connect to each other [81, 93].

Clustering coefficient and motifs:

Complex networks, in clear deviation from the behaviour of random graphs, show the property of transitivity, or clustering. If the nearest neighbours of a node are also directly connected to each other they form a cluster (i.e. if there are three vertices A,B,C, that are neighbours and each of these are connected to each other, they form a cluster).The clustering coefficient quantifies the number of connections that exist between the nearest neighbours of a node as a proportion of the maximum number of possible connections [81, 94].

$$C = \frac{\text{number of triangles in the network}}{\text{number of connected triples of vertices}} \quad (2.9)$$

Or for a vertex

$$C_i = \frac{\text{number of triangles connected to vertex i}}{\text{number of triples centered on vertex i}} \quad (2.10)$$

This leads to

$$C = \frac{1}{N}\sum_i C_i \quad (2.11)$$

Random networks usually exhibit low clustering, whereas complex networks tend to demonstrate high clustering, which is often associated with high local efficiency of information transfer and robustness.

Motifs are recurring significant patterns of interconnections noticed in the network. Thus, whereas degree distributions provide a global property, motifs enable us to comprehend local properties. Milo et al. [95] discuss a large number of network motifs which can be found in social, ecological, and technological networks, and how motifs may be used to reason about correlations between structural and functional networks. A particularly interesting application of this is the search for Ego networks in software systems to detect potential post release defects, used by Zimmermann and Nagappan in [96].

<u>Path length and efficiency:</u>

Path length is the minimum number of edges that must be traversed to go from one node to another. Random and complex networks have short mean path lengths (high global efficiency of parallel information transfers), whereas regular lattices have long mean path lengths. Efficiency is inversely related to path length, but is numerically easier to use in estimation of topological distances between elements of disconnected graphs.

If two vertices are i and j, then geodesic distance between i and j is d_{ij} if a path exists between i and j. $d_{ij} = \begin{cases} 0\,;\ if\ (i = j) \\ 0 < n < N; if\ (i \neq j) \\ \infty\ if\ no\ path\ exists\ between\ i\ ,j \end{cases}$

$$where\ \ n\ is\ an\ integer, N\ is\ number\ of\ vertices\ in\ the\ network$$

The average geodesic distance is given by

$$l = \frac{1}{N(N-1)} \sum_{i \neq j} d_{ij} \quad (2.12)$$

This is also known as E or global efficiency, where the reciprocal of E is the harmonic mean of global geodesic distances is $h = \frac{1}{E}$. This measure is often used in studies of information flow in the network [90].

Connection density or cost:

Connection density is the actual number of edges in the graph as a proportion of the total number of possible edges, and is the simplest estimator of the physical cost, such as the energy or other resource requirements of a network.

Hubs, edge, centrality, and robustness:

Hubs are vertices or nodes with a high number of incoming and outgoing links, and also vertices which fall on a large number of the shortest paths between two other vertices of the network. Thus, a hub has a central position and has relatively more influence in comparison to other nodes, in the context of interactions within the network. Interactions may be information flow, traffic flow, etc. Bullmore [81], discussing animal brain network modelling, states "The importance of an individual node to network efficiency can be assessed by deleting it and estimating the efficiency of the 'lesioned' network. Robustness refers either to the structural integrity of the network following deletion of nodes or edges, or to the effects of perturbations on local or global network states."

It is possible to quantify the importance of a hub or edge of a network by computing its centrality. Many authors have proposed different ways to compute the centrality of nodes and edges of graphs. In this review, two ways are presented: one; to compute vertex centrality, and another; to compute the centrality of an edge in the graph. A wider selection of these can be found in [90, 97, 98] [99].

The betweenness centrality of a vertex or edge u is given by

$$\mathrm{B_u} = \sum_{\mathrm{ij}} \frac{\sigma(\mathrm{i,u,j})}{\sigma(\mathrm{i,j})}, \quad (2.12)$$

Where $\sigma(i,u,j)$ is the number of shortest paths between i and j that passes or contains vertex or edge u, and $\sigma(i,j)$ is the total number of shortest paths between i and j, and the sum is the overall pairs of distinct vertices i, j.

Koschützki et al. [97], in discussing shortest paths, provide the following equation to compute Vertex Stress Centrality related to communication networks.

$C_s(u) = \sum_{i \neq u \in V} \sum_{j \neq u \in V} \sigma_{ij}(u)$ (2.12)

where u, i and j are vertices, V is the set of vertices and $\sigma_{ij}(u)$ denotes the number of shortest paths between j and j that passes through u.

The edge stress centrality is given by:

$C_s(\varepsilon) = \sum_{i \in V} \sum_{j \in V} \sigma_{ij}(\varepsilon)$ (2.13)

where i and j are vertices of the graph, V is the set of vertices, ε is an edge of the graph, and $\sigma_{ij}(\varepsilon)$ is the number of shortest paths between i and j that passes through edge ε.

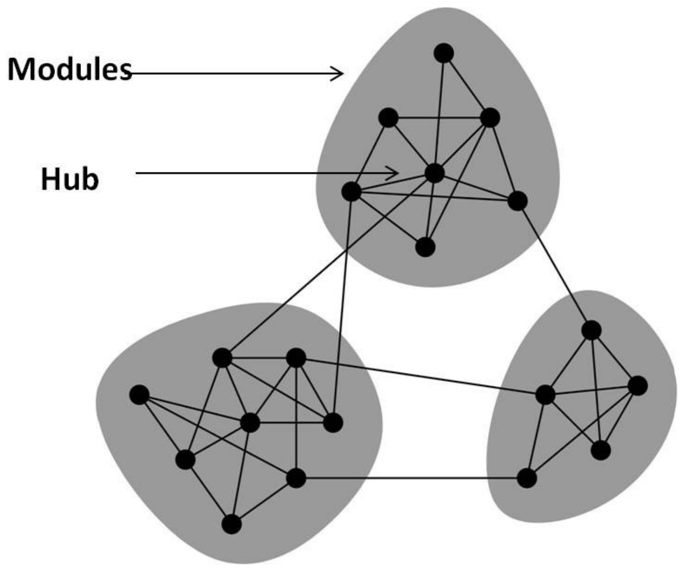


Figure 2.9 Illustration of graph modularity with 3 clusters, adopted from [100].

Modularity:

The concept of modules arises from clusters of nodes with a central hub, where these clusters appear like islands in the graph, with one or more inter-cluster links. Analysis of graphs to detect communities of nodes, motifs, or just groups of nodes that belong together, is quite common and has application in the study of social science networks [100] and biological science networks [101]. This concept is illustrated in Figure 2.9, taken from [100], and modified with labels to annotate cluster and hub.

2.3.3 Use of Complex network modelling

Complex network modelling is essentially a statistical graph theoretic approach to data analysis. Network graphs provide a domain independent skeletal mathematical framework to analyse data, based on pair-wise relationships that exist between objects of interest in the data represented as vertices or nodes and links or edges of a network graph.

The concept of functional importance of methods in classes of OO software is based on a key property of scale free networks, where these networks have a large number of vertices with few connections and network hubs that are connection rich network features. The existence of hubs makes these networks robust and yet fragile from a structural and functional integrity point of view because a random removal of vertex does not lead to network breakdown, however if a hub is removed it can lead to loss of structural integrity of the network, if not its complete collapse. The majority of literature on the application of complex network modelling and empirical network analysis measures arises from contemporary research on social systems, socio-technical systems, and computational and systems biology. Elements of research from these broad areas have led to the exploratory use of complex network modelling for software program analysis. The following sections review a selection of papers on the application of complex network models that has influenced the data analysis and empirical validation technique developed in this thesis.

Social and socio-technical systems

Travers, Milgram and Newman et al. in [80, 83, 93, 94, 98, 100, 102-105] discuss the application of complex networks analysis in the context of social science and represents the early work on social network analysis and small world networks. These works are motivated by Milgram's research [83] on social network's and acquaintance networks, that led to the concept of 6 degrees of separation. Many of these papers discuss the networks built from public databases trying to model the susceptibility of populations to the spread of virus and other infections like sexually transmitted diseases, and also provide empirical methods which are widely used to measure global and local properties of social networks. Related research in software engineering are Zimmerman's and Tosun's works [96, 106] on software defect detection that uses Ego networks and Givan's community detection [105, 107] to predict post release defects in Windows server software.

Computational and systems biology

Albert surveys the literature on scale free networks in cell biology in [87], discusses the practical implications of scale free distributions, and tries to explain the growth, evolution and robustness of biological networks. The survey illustrates, through the use of selected research, that graph representations of cellular networks and quantitative measures characterizing their topology can be extremely useful for gaining system-level insights into cellular regulation. These illustrative examples discuss how protein-protein interactions, gene regulation, and gene duplication, as well as network motifs of protein-protein interactions and topology, can be used to study local characteristics. Almaas [108] discusses the complex network analysis of protein-protein interaction network and alludes to the functional importance of important proteins, identified as critical through network analysis. Almaas argues that network graphs can be used to model protein metabolism, and that information about gene regulation and a protein's contribution to the known biological function need to be collated in order to facilitate better comprehension of how individual proteins in a protein network influence metabolism.

In [81], Bullmore and Sporns review the use of network analysis for animal neuroscience research, and discuss the challenges associated with the use of data from multiple clinical sources to build up knowledge about brain structural and functional networks. These networks are used to analyse the functional importance of topological features that represent neural elements, by using network structural perturbation techniques. These techniques are used to gain a deeper understanding of brain disease and treatment of diseases like Alzheimer’s. The review concludes that certain aspects of the organization of complex brain networks are highly conserved over different scales and types of measurement, across different species, and for functional and anatomical networks. Bullmore and Sporns also hypothesise that the small world characteristics of brain networks may be a result of the natural evolutionary tendency to achieve a high efficiency of information transfer and at low connection cost, while at the same time trying to achieve an optimal balance between functional segregation and integration that yields high complexity dynamics. The review concludes that this hypothesis needs to be explored to gain a better understanding of brain networks.

Complex networks and program analysis

Since 2002, approximately 18 research papers have been published on software engineering that uses complex network based modelling to reason about software systems phenomena using static analysis techniques. These papers explore the applicability [109], [110] of complex network based empirical models to propose new OOD metrics [111], to study software evolution and stability from a structural point of view [64], and to characterize software defect distribution [112]. There are two main differences between the key papers of interest presented in this review and the approach presented in this thesis. First, the research presented in the reviewed papers does not attempt to explain how the complex networks modelling space, and its properties and measures, can be logically mapped to the software space, such that network graph models can be used as a surrogate for software models. Second, all the empirical complex network models originate primarily from physical and life sciences research, which (however successful) does not guarantee that empirical network measures apply to models built from software program data.

In [110], Myer looks at a software system as a complex network inspired by similar studies on biological networks and OO collaboration diagrams [113]. Myer's approach to software network graphs considers edge directionality to account for the hierarchical organisation of software systems. In [110] the author also discusses software networks in light of exponential and power law connectivity distribution and empirical verification, that verifies the hypothesis that OO software may be modelled using power law scale free networks. The author also discusses OO collaboration graphs in light of network concepts like assortativity, redundancy, degeneracy, and robustness of software networks generated using static analysis techniques. The key finding in the case of procedural systems included in the study was a lack of cluster formation in the collaboration networks representing these software systems. A related research is [114] by Wheeldon and Counsell, which discusses the power law characteristics of class relationships in OO software and verifies that power law distributions exist in object oriented class relationships and attempts to correlate the findings to the C&K coupling metrics discussed in [40].

Jenkins and Kirk [64] investigate the stability and evolution of the structure of consecutive versions of a series of software architecture graphs through complex networks analysis. Comparison, between scale free distribution and second order phase transition, leads to the proposal of a new design metric (I_{cc}) to quantify the instability of the structure of software as it evolves. The metric developed is in terms of an understanding of its scale-free characteristics overlaid on the instability metric I, proposed by Martin [62].

I_{cc} refers to the instability of the entire software class-class graph and its value ranges from 1 to 0, with 1 being maximally unstable and 0 being stable. This metric is based on degree distribution, and is a measure of the maintainability of software as it evolves. In the view of Jenkins and Kirk, evolution of hub-like structures point to bad programming, but it cannot be entirely avoided in practice. The metric proposed is based on scale free networks and can be used to quantify maintainability at various levels of granularity, like classes, packages, and libraries. Valverde et al. [109] discuss organic evolution of software and how this leads to large scale software gaining small world network and scale free characteristics. The notion explored is that as evolution occurs, software attains fundamentally non optimal design due to complexity in aggregation of different elements. The study analyses software based on the largest component using an undirected network.

Concas et al. [111] present research that studies connectivity distribution models of a large number of properties of OO software. The research is exploratory in nature and aims to investigate the statistical distribution of C&K metrics [40] for the whole software system, and to reason about these metrics in the context of degree distributions found by computing in and out degree at the class level, using a directed network as a representation of the software graph. Murgia [112] tries to characterise defect distribution within a system and concludes that it roughly follows Pareto distribution – a form of power law. Murgia also reports the finding of correlations among social network analysis (SNA) metrics and bugs that are generally comparable with C&K metrics, and reports that SNA - EGO metrics [105] show the strongest correlation.

In [115], Zheng et al. perform an empirical analysis of the Gentoo network, and analyse network properties including degree distribution, sparsity, clustering coefficient, degree growth rate, and node growth. In [96], Zimmermann and Nagappan describe defect prediction in post release Microsoft 2008 server software, based on Social Networks Analysis (SNA) metrics (proposed by Newman and Givan in [105]). A directed binary dependency network was built from escrow or master copy of binaries of Microsoft Server 2008 server software network. It was found that the SNA based approach has

10% higher recall than traditional complexity metrics based defect prediction approaches used in Microsoft.

The common characteristics of these studies is that they primarily explore evolution and fault tolerance aspects of complex networks, apart from macro and micro level properties arising out of network topology [116]. This topic is still in its infancy therefore little empirical validation exists, and how it can be used is still a research topic. As a result, there is a difference of opinion in the literature as to when directed or undirected network representation should be used. Most of the research in this field is exploratory in nature and few publications report clear conclusions. Despite significant research efforts, it appears that the application of complex networks in software engineering has not made significant progress beyond the construction of software networks, and exploration of what network properties might mean when interpreted in the context of object oriented software. A common problem associated with program analysis is the computational and data storage costs associated with performing the data analysis. In this thesis, a part of the data analysis required the use of cloud computing resources. The basics of cloud computing, related to experiment process optimisation, are discussed in the following section.

2.4 Use of the Cloud

Cloud computing services do not currently have a defined standard, primarily because they are still evolving. The ideas behind cloud computing can be traced back to grid computing. Grid computing developed out of research on meta computing [117], high performance computing [118], and high throughput computing [119]. The motivation was to provide secure, reliable resource sharing to support computational science. The concept of grid was presented by Foster and Kesselman in [120] and the subsequent development of Globus Middleware [121] can together be considered as a de-facto standard for the grid. National research initiatives led to development of a grid service, where notable examples include the National Grid Service (NGS) [122] in UK and TeraGrid [123] in USA, among others. The research on grid infrastructure setup and the use of grid computing has had a wide focus, which included work on facilitating usability of the infrastructure. This aspect focused on security and access. An interface similar to Web Service was proposed, named Grid Service. However, it proved complicated and access was predominantly based on the use of secure shell (SSH), or a web interface embedded with SSH console. Grid software infrastructure is predominantly based on Unix/Linux, and did not use virtualization. Major grid services use Globus based software infrastructure, which aids interoperability and resource sharing. However, to the user the grid resources present a set configuration. Grid operations are mostly sponsored by research funding, and operation is non-commercial in nature.

Cloud computing is primarily enterprise driven, and services are offered on a commercial basis. Cloud infrastructure and services differ widely, and there is no explicit support for interoperability. Cloud resources are highly flexible in comparison to grid resources, enabling users to adopt the base infrastructure for a wide range of computing end use, by virtue of the use of virtualization [124, 125] . The user interface of most cloud services is web service based, and interactive access is possible using platform specific access technology, like SSH on Linux/Unix platform, or remote desktop on Windows platform. Figure 2.10 presents a schematic diagram of the cloud infrastructure setup. The aim is to be vendor independent. However, for illustrative purposes, some Amazon cloud service names have been used. Cloud service providers currently maintain large facilities for housing computers that can be used for computation and storage. The facilities are usually provisioned with a fast communication network, computers that front-end and back-end software based services. The services may be web services based interface facilitating user interaction with the services. The services may deal with account setup, management, and using cloud based resources. Amazon, Microsoft, Google, Rackspace, and Salesforce are all commercial cloud service providers. Cloud services use the utility model [126, 127] of charging for use of the services. In this model, users pay for resources used, and there is no standing charge.

The current interest in cloud computing infrastructure, platforms, services, and applications is partly driven by marketing ‘hype’, and partly by the genuine hope that this technological innovation can help to address issues in large scale computing, particularly that of access to resources on demand. In software testing, the often quoted statistic is that software testing is resource intensive and can account for between 40% - 70% of the cost of software through its life cycle. Most of this cost is incurred in the post construction, production use, and maintenance phases. The main issues are how much to test (test sufficiency), how and where to direct testing resources (test prioritization) in order to achieve maximum reliability, and consequently achieve high assurance. The latter question forms the central theme of program comprehension research. The issue of resource is determined by decisions on test sufficiency and test prioritization factors, in addition to test objectives, and the need for test environment fidelity to target use environment. In some cases the test resource is also determined by how the software is operated. That is, whether the software is primarily graphical user interface (GUI) driven, command line driven with or without scripting, or both.

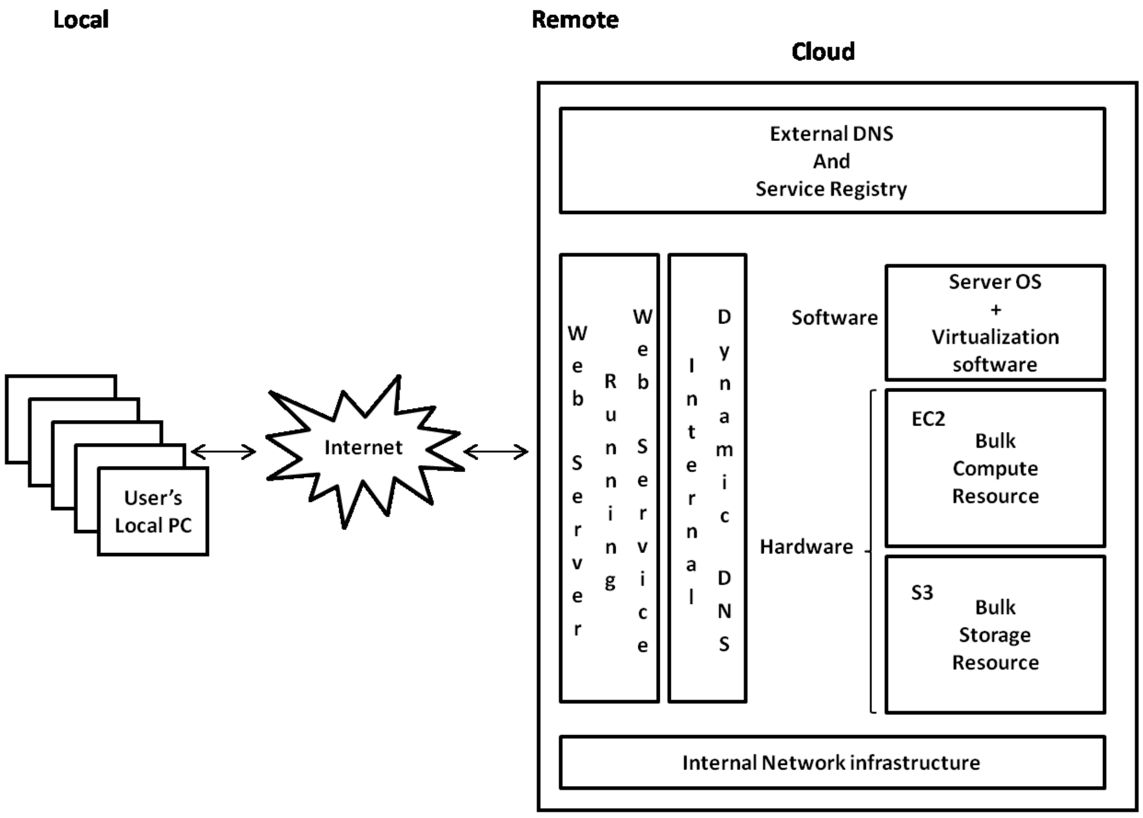


Figure 2.10 Conceptual diagram of a cloud service setup on Amazon.

Most standard software testing today – regardless of whether its purpose is system integration, regression testing based on a set of unit tests, or performance testing – relies on predetermined workflows that afford a degree of automation to the test process. In most cases, the testing infrastructure is captive in nature, with a significant degree of control over how the infrastructure is managed and operated. Early research on software testing in the cloud [128] explored how software testing workflows need to be adapted to work in the cloud. Riungu [128] also discusses the research viewpoint on what considerations may form the basis on which a decision may be formed to migrate testing to cloud. Riungu [128] also presents some examples that deal with testing as a service (TaaS), where many of these services are built on top of the Amazon cloud platform, and target small and medium enterprise involved in software application testing. These services facilitate migration of an existing testing process that uses a captive infrastructure to use a cloud infrastructure; for example [129].

In this research, the use of cloud technology has been mainly to support large scale mutant test build, task farmed data processing, and large volume data storage, which was not possible to achieve with existing captive resources available locally and that had been allocated to the research.

Chapter 3. Review of Dynamic Analysis Literature

3.1 Introduction

This research is focussed primarily on the verification of empirical measures of complex network models, when used in dynamic analysis of large software systems to support prioritization of program comprehension and software testing. Chapter 2 of this thesis discussed the essential background information on software development life cycle and static (program) analysis techniques, developed to support program comprehension and software testing of object oriented (OO) software. The key problems that need to be overcome to enable analysis of complex real world systems, including software systems, in an efficient manner are how to manage the size and complexity of the data that is produced in an effort to study these systems. As discussed in Chapter 2, complex network models and measures provide a domain-independent skeletal framework to efficiently analyse large and complex data. In the following sections of this chapter a review of literature on dynamic analysis and application of complex network analysis in dynamic analysis of OO software is presented.

Computer science and software engineering research into the prioritization of software engineering process tasks has used predominantly static analysis techniques. The bias in choice when adopting static analysis is primarily due to the lack of sufficiently developed dynamic analysis tools to undertake dynamic analysis research in a cost effective manner. Application of complex network modelling in software analysis is relatively rare in comparison to other more established statistical techniques of data analysis, primarily because these techniques are new. However, complex network based modelling and data analysis techniques are rapidly gaining in popularity, driven primarily by successful application of the research in the study of socio-technical systems and in system level studies in the life sciences.

3.2 Dynamic analysis

The intuitive notion of dynamic analysis is program analysis that involves execution of the program. A computer program is a practical realization of the universal machine [130, 131] that makes a computer system behave in a certain way, based on the sequence of commands executed. These commands (typically machine instructions) can be moving data from one memory store to another or applying transformations to the data, such that some input data may be processed into some desired output. In the case of object oriented (OO) software written in higher level languages like Java or C++, classes and methods are compiled into machine instructions or byte-code. Although it is possible to work at the low level of machine instructions, it is typically avoided for reasons of efficiency. In this thesis, the dynamic analysis technique relies on the standard approach of working at the level of source code and the compiled program.

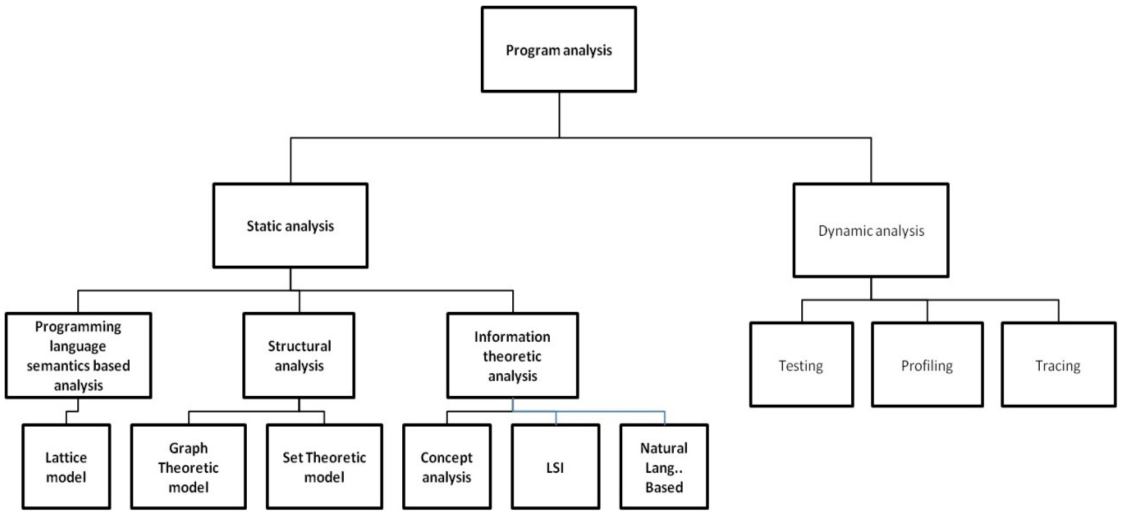


Figure 3.1 Schematic illustration of program analysis based on techniques employed.

Figure 3.1 presents an illustration of program analysis techniques (reproduced from Figure 2.1). The right branch of the tree presents three broad dynamic analysis techniques: testing, profiling, and tracing. The data collection from empirical experiments used in this thesis relies primarily on profiling and tracing, while the verification of functional importance utilises the software testing techniques of dynamic analysis. The following definitions captures the rationale behind use of the dynamic analysis techniques used in the experiments in this thesis. The IEEE SWEBOK [132] definition mentions program behaviour a key concept used in the verification of functional importance and Ernst [36] mentions the testing and profiling as standard techniques used to carry out dynamic analysis. These definitions considered with Glen's definition of profiling [133] justifies the motivation behind use of the profiling technique used to gather runtime data in this research.

The IEEE SWEBOK [132] defines software testing as follows: "Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior."

Ernst [36] states “Dynamic analysis operates by executing a program and observing the executions. Testing and profiling are standard dynamic analyses. Dynamic analysis is precise because no approximation or abstraction need be done: the analysis can examine the actual, exact run-time behavior of the program.”

Glen [133] defines profiling as follows: "A program profile attributes run-time costs to portions of a program’s execution. Profiles can direct a programmer’s attention to algorithmic bottlenecks or inefficient code, and can focus compiler optimizations on the parts of a program that offer the largest potential for improvement. Profiles also provide a compact summary of a program’s execution, which forms the basis for program coverage testing and other software engineering tasks."

Ball [134] discusses two approaches to dynamic analysis based on profiling: frequency spectrum analysis (FSA) and concept coverage analysis (CCA). In FSA, program runtime information, like invocation frequency of program entities, is used to facilitate the understanding of performance of program sections, and also help with performance optimisation related activity. In CCA, the concept analysis technique is applied to test coverage data to help compute dynamic analogues to static control flow relationships, such as domination and post domination regions. Comparison of this dynamic data and static program data can be used as a prioritisation aid to optimise testing efforts.

It is important to notice that dynamic analysis is only possible to perform at the coding phase in the form of unit tests, at the pre-release stage of the software life/process cycle, and at the maintenance phase, because an essential requirement of dynamic analysis is the existence of an executable program. It is true that at the coding phase the complete system may not be available, and therefore the unit test is performed at the level of modules or units on a test rig that usually tests software units based on inputs and outputs. A common way to verify that software behaves as expected is by interacting with and observing its manifestation when it is executed. That is, when the software responds to stimuli or interaction. We perceive the program’s behaviour based on its response. Theoretically, a set of stimuli can produce many responses from an executing program, especially from a large software system of Lehman's P and E type [19, 135]. In this context, Milner's and Fokkink's discussion [136, 137] on a formal methods approach to modelling sequential and parallel behaviour of modern computer systems at the process level explains why static analysis [138] and formal methods based modelling and verification can become un-scalable. Namely, this is due to the potential for state space explosion, due to the almost infinite set of possible inputs, execution paths, and outputs that result from modelling Lehman's Type A (P,E) systems. In static analysis, state space modelling allows us to reason approximately, but soundly, about software phenomenon during runtime. In dynamic analysis, if we consider the state space model, the data collected is precise [36], [134] and applies to a particular exercised execution scenario, as described by a specific set of interactions with the executing program.

Howden [139, 140] discusses the issues of functional testing: how it is performed and the selection of input values determined from the domain (i.e. verification of what the function is designed to compute, the intended range of inputs, the expected path the function should take, and the expected outputs). The discussion is close to what is sought to be achieved through unit testing or mutation testing. Yue and Harman survey the concepts and techniques of mutation testing in [141], discuss various approaches to mutation testing, and mention the cost intensive nature of such testing. The experimental work presented in this thesis uses techniques very closely related to those discussed by Howden [139, 140] and Yue and Harman [141], but for OO software. A key challenge for dynamic analysis is data collection, data exploration, and visualization [142, 143]. In the following sections a discussion of the key dynamic analysis literature is presented.

3.2.1 Software testing and profiling

Meyer [144] discusses seven best-practice based principles of software testing, covering test definition, tests versus software specifications, regression testing, contracts as oracles, manual and automatic test cases, empirical assessment of testing strategies, and assessment criteria. The core principles are that software must be tested with the aim to make it fail, but software tests cannot be a replacement for system specifications. Ideally, tests must consist of both automatically or tool generated test code, and also manually generated test code, and every failed regression test must lead to a new test case with a clear test-end indicator. On the subject of test strategies, Meyer's principle advocates empirical assessment, and the assessment criteria should be reliably based on measurement of time bound fault detection. These ideas fit neatly with the discussion of testability and test sensitivity by Voas [145-147] and the discussion of the benefits of agile process and test driven development processes by Talby et al.'s [26], which are complementary to Meyer's principles [144]. However, Jackson presents a contrasting view [28, 148], in which he points to the futility of large scale testing and recommends user driven testing, like user acceptance testing, as an effective testing scheme. Yue and Harman discuss the wide range of techniques employed to carry out mutation testing [141]. Some of these techniques have also been used to generate mutants in empirical experiments on software testability and test sensitivity by Voas in [145-147]. The problem with software mutation testing is that it is expensive and requires enormous human effort [141]. The dynamic analysis techniques adopted in the experimental part of this thesis use a form of user acceptance testing for data collection, and mutation testing for verification of predictions of functional importance.

Data collection in dynamic analysis research is a key bottleneck in this context, where Waddington et al. [149] discuss a wide range of tools that facilitate the profiling of multi-threaded software systems and techniques to collect dynamic analysis data. The discussion on dynamic and binary instrumentation (DBI) frameworks is particularly insightful and shows the growing popularity of virtual machine based DBI frameworks. Waddington et al. [149] also discusses the performance bottleneck that is typical in profiling large scale software and provides a comparative overview of the tools, although the paper does not report empirical results. The main thrust of the discussion is to demonstrate effects like the slowdown of the application and problems with response time under two different execution conditions: with instrumentation of the executable program, and without instrumentation of the executable program. The results are then used in discussion of a scaled-up scenario where the actual application being profiled can be many times larger. The applications used for performance experiments are common benchmark codes used in testing both software tools and platform performance, especially those connected with performance profiling, such as LU DECOMP and Huffman.

3.2.2 Data collection

In the context of Waddington et al.'s discussion [149] on data collection bottlenecks and the use of DBI, a contrasting technique is static instrumentation to the source code. Geimer et al. [150] discuss the TAU framework that conducts static instrumentation of program code and is primarily used in High performance computing (HPC) environments. The paper [150] investigates the basic constructs required to specify user-defined method entry/exit instrumentation in a generic fashion. It also investigates generic building blocks that can be applied as instrumentation, and features like keyword substitution, which are found useful in taking advantage of participant knowledge at the time of instrumentation. The idea behind key word substitution is to build in an intelligent instrumentor that is aware of build tools.

Cain et al. [151] discuss improvement to the instrumentation mechanism in Paradyne PC (performance consultant) profiler software, where selective dynamic instrumentation is implemented by incorporating call graph searches that takes into account the path being taken as the program executes. This discussion of performance improvements, due to use of this new instrumentation technique, has been illustrated by using 6 example programs that are procedural in construction and written in C ad Fortran. The reason behind improved performance is that the scheme searches for valid paths to instrument, rather than using blanket instrumentation. The main reason behind the slow performance of dynamically instrumented applications at runtime is that more time is spent on instrumentation tasks that are never used, than on actual computation related to the application.

Nethercote and Seward [152] discuss the Valgrind tool, which can be used to carry out dynamic binary instrumentation/introspection of a binary program. The instrumentation scheme is based on intermediate representation, and is known to be a heavyweight technique. This technique is known as heavyweight because intermediate representation makes the scheme more flexible for use with different compilers, architectures, and software tools, but use of the Valgrind scheme requires significant programming effort and the technique is also computationally intensive because of the translation related load associated with the use of intermediate representation. Tools like Valgrind provide a framework upon which user profiling tools can be built. However, such endeavours require significant effort. Intermediate format use requires mapping from microcode to intermediate format, then instrumentation occurs in the intermediate format level, which is turned back into microcode and executed, essentially requiring two stages of translation.

Hundt [153] introduces the HP Caliper framework for program analysis. The framework is a probe-based dynamic instrumentation framework that relies on a virtual machine scheme, as opposed to the intermediate representation scheme used by Nethercote and Seward [152] in Valgrind. The HP Caliper framework tool is based on the ATOM library, which provides an API for dynamic instrumentation. Skaletsky et al. and Moshe [154, 155] introduce and discuss how Intel's Pin library [156] based tools can be used for tracing and dynamic analysis of large binary applications. Pin [156] is a VM based, profiler directed instrumentation framework. Dynamic instrumentation typically requires significant effort to implement, and often the approach has been to write very low-level assembly language code to achieve this. The problem frequently encountered is that instruments became application specific and not easy to use for different applications on the same platform. The other major problem has been in the performance of the instrumented application, due to the overhead associated with the instrumentation process. Pin tool, which is in part inspired by ATOM, is a framework for dynamic binary instrumentation. The tool provides an application programming interface (API) for high level programming languages, primarily C and C++. The API abstracts some of the low level architecture details that earlier approaches had to deal with, making it possible to build generic tools that can be used to build profilers, tracing tools, and so on. These frameworks take advantage of compilers to achieve optimisation of the instrumentation process during runtime. The common low level intermediate format used by the C, C++, and Fortran programming languages has been utilised in the VM layer with the tool, such that application binaries written in these languages for x86-32, x86-64 and other architectures based on x86 can use one single instrumentation tool. DynamoRIO [157] is another tool that provides facilities similar to Pin and ATOM.

Binder et al. [158] present a new framework for dynamic byte code instrumentation in Java and FERRARI, based on BCEL, and discuss both static and dynamic binary instrumentation of java applications. Tanter et al. [159] discuss Aspect-oriented programming (AOP). AOP based techniques and tools[160] are popular and used widely in dynamic analysis [142] and software testing and profiling to collect runtime data, especially for software written in Java [161]. A major attraction is the use of a compiler based weaving technology with a high-level interface (i.e. API) that does not typically require programmers to work at the level of microcode.

Couture et al. [162] reviews a wide range of tools available for static and dynamic instrumentation that are currently available, primarily for applications written in Java. The data collection techniques used in our research utilised static Java byte code instrumentation Eclipse Probekit [163] which is similar to the scheme discussed by Geimer et al. [150], and the Eclipse TPTP [164] that employs DBI, similar to the scheme discussed by Binder et al. [158]. In the case of C++ programs used in our research, the instrumentation scheme used was built on Microsoft Debugger technologies that are similar to techniques discussed by Hundt [153] and Zhao et al. [165]. In the case of Zhao et al. [165], the technique discussed uses the GNU Debugger

[166, 167]. The key difference with [165] is in the use of break points. The techniques that are fundamentally different in approach, such as Nethercote and Seward [152] and Cain et al. [151], were both tried but were found to suffer from significant scalability problems discussed in [149].

3.2.3 Object oriented design quality

Dynamic analysis based research concerning object oriented design (OOD) quality is limited, primarily because of the lack of robust tools to support data collection and the popularity of static analysis based metrics. In this context, Arisholm et al. [35] propose a dynamic coupling measure, formal definitions, and a framework to define external/external coupling measures. Arisholm et al. [35] describes a prototype data collection and analysis tool to perform dynamic coupling measurements, and presents empirical validation to demonstrate that dynamic coupling can be used as a change proneness predictor. The dynamic coupling metrics suite developed by Arisholm et al. [35] is similar to static coupling metrics suite proposed by Briand [41, 42]. Chhabra and Gupta [168] present a comparative survey of important dynamic analysis metrics. The key findings are that despite being found to be more precise than metrics computed by static analysis, little research exists because of the expense of calculating these metrics from dynamic analysis data. The survey [168] reports that pseudo-dynamic metrics is a topic that has not received much research interest, and suggests that the use of pseudo-dynamic metrics may be beneficial because these may be cheaper to compute and comparable to metrics extracted by purely dynamic means.

3.2.4 Program comprehension

Zaidman et al. [169] propose a technique that uses web mining principles for uncovering important classes in a system's architecture, using graph theoretic page ranking techniques and the HITS algorithm [170]. This technique is also known as the web mining technique, popularised by Brin and Page [171]. Algorithms of this class have also been discussed by Newman [102] while discussing power laws. In [169], the HITS algorithm is used to discover rank classes, based on analysis of the object-object interaction network built from execution traces. The aim of this work is to support program comprehension by providing developers with a starting point for the comprehension process. The premise is that the objects (instances of classes) which are ranked highly by using the HITS analysis are likely to indicate important objects, and consequently classes, in the software. The authors also mention the CBO coupling between objects metric discussed by C&K [40], and present this ranking as essentially uncovering highly coupled classes. Zaidman et al..'s work in [169] is very similar to the work on the identification of functionally important methods in OO software. A crucial difference is that no effort is made to associate the network analysis model to the software engineering domain in terms of class collaboration networks, as in [110] and [172]. In addition, the data collection technique used to collect the class level dynamic interaction data is not described.

Cornelissen et al. [173, 174] discuss how large volumes of dynamic analysis data can be visualised to support program comprehension through dynamic analysis. The technique adopted to collect execution trace data is based on use of the tracing aspect, written in Java using AspectJ [160, 161]. The visualization scheme organises hierarchical execution trace data, collecting class level interactions into circular bundles of entities. These entities are expandable and collapsible, based on how the user interacts with the visualisation tool. All the entities are displayed along the circumference of a circle in the visualisation tool. Intra-elemental communication events (i.e. message exchanges) are denoted by lines connecting the elements, with line thickness denoting the intensity of communication. The research presents a program dynamic analysis data visualisation tool to support program comprehension, controlled experiments to validate the scheme of visualization, and also validation of the scheme in the context of Shneiderman's [175] seven criteria for user interface design.

Lienhard et al. [176] present an approach to track and visualise object flow through its lifetime, to understand software behaviour during runtime. Object references are treated as first class entities for tracking object flows as the system executes. The approach presented by Lienhard et al. [176] contrasts with dynamic analysis techniques dominated by a message passing view of the system presented by Cornelissen et al. [173, 174]. In the message passing view, it is primarily the control flow that is being analysed; in the object centric view, it is the data flow that is being analysed. The authors have implemented their object tracking system for a Smalltalk system and a web application, but not for Java. It may be easier to track objects in some languages (like Smalltalk) than in others, particularly when the languages are pseudo-object oriented (like Java that allows class members to be both native types and objects). The research is presented as visualisation of object flows to support program comprehension, and the authors claim that it provides complementary information to message passing dynamic analysis.

Bohnet et al. [177] discuss a concept and feature visualisation technique that facilitates pruning and scalable visual exploration of large traces to support program comprehension. They discuss how pruning of large information traces is achieved in order to facilitate scalable visualisation. Similar and repeated function call sections of the trace are collapsed into single elements by computing the call fingerprint function, which is represented as a bit vector. The bit vectors are analysed for similarity, which leads to pruning.

Singer and Kirkham [178] discuss dynamic concept analysis, visualisation, profiling and the Java virtual machine's garbage collection. They also discuss the use of aspect oriented programming [160, 161] and program and behaviour comprehension. The key idea discussed in [178] is feedback directed concept assignment, where the user assigns a concept to parts of code based on tool driven code inspection and subsequently executes it to verify and update the assignment of concept to code. Singer and Kirkham [178] suggest that similar feedback directed optimisation of a concept assignment can, in future, be implemented in a profiler tool to support interactive concept analysis. The authors state that the user annotation based approach demonstrated in the paper is currently not possible using standard tools, despite compiled code retaining a significant amount of information. The need exists to develop techniques for profiler directed optimisation; in order to perform an automated concept assignment based on the annotated information that already exists in the compiled code. The automated concept assignment can then be refined by the tool based on user feedback.

Hamou-Lhadj et al. [179] and Hamou-Lhadj and Lethbridge [180] present a semi-automatic approach for summarising the content of large execution traces. The summarisation is achieved by using Use Class Modelling (UCM), as opposed to UML. UCM provides a compact way to represent method call graphs generated from execution traces. Another aspect is to remove contributions in the trace, due to utility methods or helper methods, by using an algorithm that uses fan-in, fan-out analysis [33], and manual adjustment of thresholds which help to filter out utility methods. It is widely recognized that large traces are difficult to analyse with information (that can be considered as noise) obscuring the core information required for program comprehension. The research is reported as initial work on small systems that has yielded encouraging results. Hence, further validation of the technique is required on larger systems.

Kuhn and Greevy [181] discuss the issue of dynamic trace analysis posed by size and complexity. They propose the use of signal processing techniques to treat trace entries as events and look at the coarse grain view while preserving essential information, while reducing storage and computational cost at the same time. They also present a technique for visualising trace information as a time-series of events extracted from the trace using a signal processing technique. The research represents an effort at trace abstraction, by use of data mining to support program comprehension.

Dynamic analysis research, used to support program comprehension, suffers from the challenges posed by large volumes of complex data that is generated and the lack of robust data collection tools. AspectJ or profiler based tools appear to have been used in most of the papers considered [173, 174], [179], [180] and [181], presenting the first significant difference with the data collection technique employed in this thesis. The common thread is the collection of full traces. In this respect, this thesis draws significant motivation from [173, 174].

3.2.5 Complex networks based modelling

Potanin et al. [172] argue that object interaction network graphs of object oriented software display no characteristic scale. They present results which indicate that in-degree and out-degree object reference distribution has scale free power law distribution. The research actually corroborates the results discussed by Myers [110] in the context of static analysis. Potanin et al. [172] uses custom stack tracking for data collection and tracking objects, which is a technique that can be considered to provide better performance when compared to full scenario tracing, because the size of the data collected can be significantly reduced depending on requirement. However, the technique has a significant potential for error, unlike the Java Eclipse TPTP [164, 182, 183] based approach employed in this thesis. A significant difference in the analysis approach with this thesis is the use of directed network measures. However, the author does not explicitly mention which network measures were used, unlike this thesis, thereby making the empirical validation questionable.

3.3 Mixed mode analysis

It should be increasingly apparent that static and dynamic analyses are rarely mutually exclusive in practice. Ernst [36] advocates the need for "hybrid analysis that combines static and dynamic analyses. Such an analysis would sacrifice a small amount of the soundness of static analysis and a small amount of the accuracy of dynamic analysis, to obtain new techniques whose properties are better-suited for particular uses than either purely static or purely dynamic analyses." Jackson and Rinard in [28, 148] seem to echo a similar thought when discussing the dependability of software systems, and Meyer [144] states that in some cases random testing yields similar results in comparison to complicated and testing schemes, leading to his seven principles. One can interpret several techniques discussed in the literature as applicable to either mode: they are not purely static, nor are they purely dynamic. There is still no universal theory that applies to all modes of software analysis, although that would be ideal according to Bertolino [184]. In this section, the discussion presents literature that uses both static and dynamic analysis techniques to support software testing and program comprehension.

3.3.1 Software testing and profiling

Taitelbaum et al. [185] introduce the concept of synergistic assurance and propose a technique that demonstrates its implementation in real programs, by mixing formal verification and dynamic verification – almost like contract driven testing. Rothermel et al. [186] discuss several techniques for using test execution information to prioritise test cases for regression testing, including: 1) techniques that order test cases based on their total coverage of code components, 2) techniques that order test cases based on their coverage of code components not previously covered, and 3) techniques that order test cases based on their estimated ability to reveal faults in the code components that they cover. The test suites considered in the empirical analysis were untreated, randomly ordered, and optimally ordered. The results indicate that prioritisation improved the rate of fault detection, even in the case of the least sophisticated prioritisation scheme. It was also found that while techniques based on the fault exposing potential outperformed other techniques in some cases, these techniques are generally not cost effective and may require further research to perfect. As a result, interest is likely to be limited to academic rather than industrial uses.

3.3.2 Data collection

Ayewah et al. [187] describes a lightweight instrumentation framework built by Microsoft that is used to collect usage data on static analysis tools, bundled with various Microsoft provided development platforms. These data are used to improve the usability of the tools and to schedule maintenance activities. The key feature of this framework is that it facilitates collection of application usage data in an unobtrusive manner from a wide range of applications, thereby enabling efforts to minimise post release defects by pre-emptive maintenance.

3.3.3 Program comprehension

Poshyvanyk et al. [188] recasts the feature location problem as a decision making problem. They use dynamic analysis (SPR), static analysis data, and an information retrieval (IR) technique (LSI) to create a new hybrid technique PROMESIR. SPR is a feature location application that uses dynamic analysis based on scenarios, and IR approaches are used for various tasks such as traceability link recovery (i.e. linking source code to documentation). PROMESIR is a tool that implements a hybrid of SRR and IR approaches. The tool has been used for bug location, by utilising complementary analysis techniques and data that SPR, and IR based on LSI, represents. PROMISIR is found to be accurate but computationally expensive.

Eisenbarth et al. [20] use static and dynamic analyses for feature location using concept analysis, and dynamic analysis to reduce the search space. First, the feature location of system components is achieved by scenario execution and generation of traces. Then, concept analysis is carried out using static analysis. Empirical validation is conducted on the Mosaic and Chimera browser software. The authors argue that partial architecture recovery using this technique can support program comprehension.

Bohnet and J. Döllner [189] present a mixed mode static/dynamic analysis based technique to visualise the call tree control flow graph of very large software (>1MLOC) along with feature location, in order to support program comprehension and maintenance tasks. The authors argue that feature location, and comprehension of how the features map to architectural components, are both important from the software re-engineering point of view making visualisation of source code features, call graph, and architectural components important. The prototype tool facilitates program understanding through visualisation.

Rajlich and Wilde [66] discuss the learning of domain concepts from a program through static and dynamic analyses. The authors discuss the importance of concept location, and decomposition of a system, based on concepts. This is an early research paper which discusses a decomposition scheme called de-location, which appears to be counting concepts in marked parts of code. Few details of the technique employed are provided, but it appears to be an interesting approach to mapping domain concepts to source code, purely based on a tagged instrumentation scheme to generate the data.

Kontogiannis et al. [190] discuss the challenges facing program comprehension in the context of software constructed using multiple languages. In single language implementations, the tasks typically revolve around uncovering the dependencies between concepts, features, and code. However, this task is more challenging in systems written in multiple languages and containing many libraries and complex inter-language interfaces. This problem has also been mentioned by Goth [3].

Voigt et al. [143] discuss large execution trace exploration and visualisation. The authors propose a data model to map an execution trace into a visualisation system; the aim being to enable efficient scalable visualisation of dynamic analysis data, and to facilitate program understanding through understanding of non-local effects of object oriented software design and construction. The work contrasts with, and also is comparable with at a certain level, the work of Cornelissen et al. [142]. Voigt et al. [143] and Cornelissen et al. [142] are comparable because both works deal with dynamic analysis program data visualisation as a program comprehension aid. The contrast arises at the level of data collection techniques adopted, with Voigt et al. [143] using profiler debugger facilities whilst Cornelissen et al. [142] relies on the use of an AOP based custom trace data collector. Based on the discussion in [149] one may conclude that the data collection approach used by Voigt et al. [143] would be more scalable.

The data collection and analysis approach of this thesis is significantly different from most of the research presented in this review that covers dynamic analysis to support software testing and program comprehension. Our research identifies closely with the scenario driven runtime trace data collection technique adopted by Cornelissen et al. [142] for software written in Java, and Voigt et al. [143] for software written in C++. The conceptual basis of our research is closest to Zaidman et al. [169], Potanin et al. [172], and Myer [110]. The review so far has presented only those papers that provide inspiration for different aspects of our research except empirical analysis, which we review in the next section.

3.4 Empirical experiment in software engineering

Software engineering experiments follow an iterative process consisting of the following four phases:;1) definition, 2) planning, 3) operation, and 4) interpretation, according to a framework proposed by Basili et al. [191]. This work [191] and Shull et al. [192] both recognise that the techniques and methods employed in software engineering experiments are likely to be characterised by diversity and guided by the context in which the experiments are conducted.

Today, with the increasing use of sophisticated models in static and dynamic analysis, coupled with a multitude of metrics, there is a need to state clearly what is sought to be measured, how accurately the models are able to measure properties of interest, and how widely applicable these measure are across a class of software systems. It is also good practice to clearly state any threats to the validity of the analysis, and the assumptions that have been made in building the models. Deeper discussion of the topic of experiments in software engineering, encompassing elements of measurement theory and statistics, is beyond the scope of this review. However, no current review should ignore the empirical experimental aspect and the trend towards evidence driven practices both in research and industry [193].

Application of measurement theory, modelling, and experiments

Fenton and Pfleeger [29, 194] and Briand et al. [59, 195] discuss the fundamental concepts of measurement theory in terms of the Empirical Relational System and Formal Relational System, and discuss how measures of software can be developed in the context of these theoretical underpinnings.

Name of Scale	**Transformation g**
Nominal Scale	Any one-to-one g
Ordinal Scale	g - Strictly increasing function
Interval Scale	$g(x) = a\,x + b$
Ratio Scale	$g(x) = ax$
Absolute Scale	$g(x) = x$

Table 3.1 Statistical scales and valid transformation that can be applied [196].

The main issues discussed in this context are scales of measures and transformations that can be applied to such measures based on its scale (see Table 3.1). The statistical methods used in data analysis are typically determined by how the data is collected, what it represents, and what is sought to be measured or modelled based on the data (see Table 3.2).

Pickard et al. [197] discuss the use of meta-analysis and vote counting techniques to conduct empirical studies in software engineering. The aim of the research is to motivate empirical software engineering experiments to follow standard techniques in a rigorous way, so that the conclusions reached can be relied upon to be widely applicable across the reported studies.

Scale Type	**Examples of appropriate statistics**	**Type of appropriate statistics**
Nominal	Mode, Frequency, Contingency Coefficient	Nonparametric Statistics
Ordinal	Median, Kendall's tau, Spearman's rho	Nonparametric Statistics
Interval	Mean Pearson's correlation	Nonparametric Statistics and parametric statistics
Ratio	Geometric Mean Coefficient of Variation	Nonparametric Statistics and parametric statistics

Table 3.2 Illustration of scale types, possible software measures and appropriate statistics [196].

El-Emam [198] discusses in detail the statistical aspects of validating software product metrics. The need for good experimental design is stressed, coupled with the choice of statistical technique. According to El-Emam, the choice of statistical technique depends on the scale type of the independent and dependent variables in the model. The author also states that there is significant scope for research on the statistical aspects and their application to analyse experimental data in software engineering. However, El-Emam warns that statistical analysis techniques need to be applied with care in empirical software engineering, because application of these techniques can often appear too restrictive and can also produce spurious results, for instance Fenton's discussion on the Goldilock's Conjecture in [194].

Kitchenham et al. [199] provide preliminary guidelines to design, prepare, conduct and analyse, interpret, and present the results of empirical software engineering experiments. There are four guidelines that relate to the interpretation of results alone; 1) define the population to which inferential statistics and predictive models apply, 2) differentiate between statistical significance and practical importance, 3) define the type of study, and 4) specify any limitation of the study. The authors propose the guidelines based on deficiencies highlighted by their survey, and in the belief that if the guidelines are followed then the deficiencies could be substantially addressed. Sjoeberg et al. [200] state that the classical method for identifying causal relationships is to conduct controlled experiments. Their paper surveys the quality of research and techniques adopted in the context of controlled experiments. They report that only 1.9% of the surveyed literature actually discussed controlled experiments. According to the authors, a major finding of the survey is: "reporting is often vague and unsystematic and there is often a lack of consistent terminology. The community needs guidelines that provide significant support on how to deal with the methodological and practical complexity of conducting and reporting high-quality, preferably realistic, software engineering experiments." This observation was also made by Kitchenham et al. [199]. In the context of this thesis, a significant part of the experimental results depend on empirical analysis. The following literature deals with the quality of empirical software engineering experiments: Fenton and Pfleeger [29, 194], El-Emam [198] and Kitchenham et al. [199], and each provide guidance on good practices to follow in the conduct and reporting of empirical software engineering experiments. The topic of statistical physics of complex networks, network analysis, and ranking techniques has not been discussed, highlighting the need for more work on this topic.

3.5 Network analysis

3.5.1 Complex networks and program analysis

In this section we begin the review with a work by Pastor-Satorras et al. [201] that discusses the growth of the world wide web and reasons for its susceptibility to directed attacks, based on the node-edge connectivity distribution and properties of the Scale Free network model[7, 202]. The remaining papers describe the key exploratory research dealing with the application of complex network models to analyse OO program data. Finally, two papers are reviewed that present the application of a complex networks inspired software test engineering technique, for defect detection in the context of a large scale industrial software production environment. These papers, along with a paper by Bullmore [81], are the key motivators behind the complex networks modelling used in this thesis.

Pastor-Satorras et al. [201] show that the internet's growth and structure have been found to have a connectivity pattern that follows the power law model. There has been significant interest in modelling the growth of the internet and also the World Wide Web (WWW). In the case of the internet, data is generated primarily using trace route program output, while for the WWW, data has been generated by web trawling. In web trawling, links on the page are taken as out-going links and followed iteratively, which provides data about the incoming and outgoing links of each webpage. This data is then analysed using network analysis, either to cluster or to rank pages using different criteria, for example, in page ranking algorithm and web search optimisation by Brin and Page [171]. Another focus of the modelling of technological networks is robustness to directed attacks and accidental failure, given the economic importance of the internet and its autonomous functioning.

Myers [110] and Potanin et al. [172] discuss how software systems also display power law network properties. While Myer's work uses static analysis data and object oriented class collaboration network [113] parsed from source code, Potanin's work [172] is complementary and uses objects (i.e. dynamic data from software executions) to confirm that software networks display scale free properties. Both works represent early curiosity-driven exploration of complex networks modelling of software/program data, possibly using strongly connected component (SCC) detection methods found in Dorogovtsev and Mendes [92].

Zimmermann and Nagappan [96] and Tosun et al. [106] apply complex networks, specifically social network analysis, to the prediction of defects in post release software. While Zimmermann and Nagappan propose the technique, Tosun validates the results by partially reproducing the study. The reason behind the partial reproduction is that Zimmermann and Nagappan [96] used Microsoft's internal development versions of Windows server software. Recent research by Murgia [112] uses software defect reports data and complex networks analysis (but not social network analysis) to explore software defect distribution patterns, and concludes that defect distribution may be characterised by Pareto distribution (i.e. 80% of defects arise in only 20% of the system). The question as to whether this can be due to programming errors, design errors, or other causes has not been discussed.

3.6 Review notes

Dynamic analysis is not new, and software engineers regularly engage in the process. The problems of object oriented software systems in forward engineering are not as acute as the problems with reverse engineering. The main problem is in mining a large volume of program data to gain an appropriate level of comprehension. Another problem is in generating or capturing dynamic program data in a reliable way. A significant number of papers discuss the prioritisation of maintenance tasks by using information gained from static analysis, while others discuss how to narrow the search space of software artefacts to investigate. A large number of works discuss defect detection or prediction, and some discuss impact analysis.

In complex network analysis, one problem is clearly the validation of empirical measures. The majority of papers appear to be using empirical methods arbitrarily, inspired by claims of success in fields for which these methods were originally defined. Data concerning social networks is relatively easier to acquire today, given both the spread of software and computers, and the popularity of the internet and WWW. Acute data collection problems and validation challenges are present in the area of animal neuroscience. It is possible to characterise brain networks as having scale free distribution, and also possible to theoretically map structural and functional networks; but it is hard to see how the results of any analysis can be verified easily in a clinical or laboratory environment without significant investment of time and money. An alternative to clinical verification and validation is to explore if software program data may be treated as a surrogate for biological data, given that software systems are often considered similarly complex. If such an approach works in validating the network analysis, determining whether conjectures about robustness or structural-functional network correlation actually hold will become significantly easier. This approach is also likely to benefit the field of software engineering.

A key problem in dynamic analysis is extraction of runtime data from multi language object oriented software systems. This is due to tools being unstable or non-existent. Another problem is the need to define and collect data collection and storage standards that apply across languages. In addition to these issues, performance of the dynamic analysis environment has also been discussed in the literature, primarily noting the cost of data collection on performance and the scalability of existing testing schemes.

In this review, a large volume of peer-reviewed literature and information published online was considered. A selection of these items, that was considered most relevant, has been reviewed in this thesis.

Chapter 4. Concept of functional importance

4.1 The concept of Functional Importance

The concept of functional importance is present by the implication of various approaches to software comprehension, testing, evolution, and software maintenance [6, 186, 203]. Naturally, finding such important elements (e.g. classes, methods) of the software is critical for appropriate test prioritisation or efficient program comprehension. Here we provide our definition of the concept of functional importance for methods.

A given functionality of the software can be mapped onto a set of usage scenarios for the software – for example by considering variations of use cases relevant for a functional requirement of the software system. A usage scenario is defined by a sequence of user operations that are expected to be executed without the user experiencing undesired software behaviours. The expected behaviours of the software are set by the requirement's specification of the software system. From the user's perspective, the usage scenario can be represented as a set of user events, including both user perceptions and actions, which are arranged in temporally ordered patterns. Thus, an event can have a duration. Note that several user events may happen fully or partly simultaneously, such that they may start and end at the same time or at different times. The user events E1 and E2 are temporally consecutive if event E1 has to terminate before the start of another event E2, and there is no intervening interval during which these two events overlap. The temporal consecutiveness of user events defines a temporal ordering graph over the set of user events corresponding to a usage scenario. An example of a temporal ordering graph representation of a partial usage scenario is shown in Figure 4.1. In Figure 4.1, the user-events represented by dashed surroundings are user perceptions, while the user events indicated by continuous surroundings are user actions. Temporal consecutiveness of user events is indicated by arrows.

From the perspective of the software system, the usage scenario is represented by a trace that consists of a set of software events that are arranged in temporally ordered patterns. The trace may include multiple threads of software events (or operations) that can be considered at different levels of granularity. It is possible to consider the software events in terms of method calls that are executed between object instances of classes (an object may call one of its own methods, and two object instances of the same class may also call each other's methods).

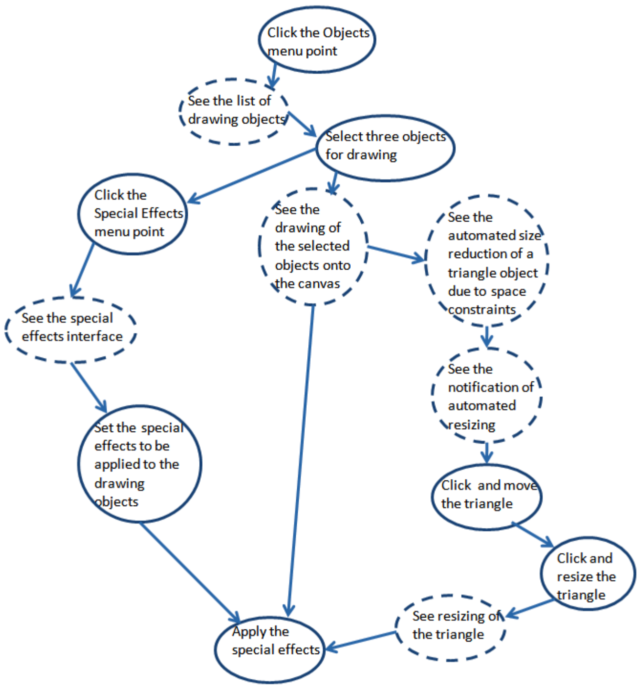

Click the Objects menu point
See the list of drawing objects
Select three objects for drawing
Click the Special Effects menu point
See the drawing of the selected objects onto the canvas
See the automated size reduction of a triangle object due to space constraints
See the special effects interface
See the notification of automated resizing
Set the special effects to be applied to the drawing objects
Click and move the triangle
Click and resize the triangle
See resizing of the triangle
Apply the special effects

Figure 4.1. Temporal ordering graph representation of a partial usage scenario from the perspective of the user.

Just as in the case of user events, several software events in the form of method calls may happen sequentially or simultaneously, and the temporal ordering of software events is determined by the inherent temporal consecutiveness relationship between these events, which can be defined in the same way as user events were defined in the preceding discussion. The temporal ordering of software events defines a temporal ordering graph of these software events, and represents the trace corresponding to the usage scenario from the perspective of the software system.

The temporally ordered graphs of user events, corresponding to a usage scenario, forms two user experience equivalence classes UC^+ and UC^-. UC^+ is the equivalence class of such graphs that correspond to the appropriate execution of the scenario that fits with the desired behaviour of the software from the user's perspective; and UC^- is the equivalence class of all other such graphs that correspond to the inappropriate execution of the usage scenario that includes undesired behaviour of the software (e.g. experiencing a crash or the inability to execute a required operation).

The desired behaviour of the software may include some unexpected behaviour of the software, if these unexpected behaviours do not affect the expected functionality of the software (e.g. a slight change in the colour shade of the background of the menu bar may be unexpected but may not affect the expected functionality of the software, because it is functionally unimportant). The undesired behaviour of the software always includes some unexpected behaviour that the software should not do if it conforms to the specification of the user requirements. We should also note that the desired behaviour of the software may depend to some extent on the user as well, and on the strictness of the definition of what desired behaviour is and what desired behaviour is not. For example, changing the background colour of a panel from light blue to light red may be regarded as acceptable and within the limits of desired behaviour by one user, while the same may be regarded as undesirable by another user. The desirable behaviour is defined as the expected behaviour according to the requirements specifications – this defines the UC^+ equivalence class. To construct an approximation of the UC^- equivalence class, we may consider all kinds of possible undesired disturbances of the user experience – of course, some of these may have no corresponding software execution trace.

The UC^+ and UC^- equivalence classes induce a corresponding equivalence class structure over the possible temporally ordered graphs of software events that correspond to the considered usage scenario: SC^+ is the equivalence class of such graphs of software events that correspond to user event graphs in UC^+; and SC^- is the equivalence class of software event graphs that correspond to user event graphs in UC^-. It is possible that more than one software event graph corresponds to a single user event graph because the software event graphs include all software events that correspond to a usage scenario, and some of these events may not have any user-observable impact on the software behaviour. Consequently in such cases, somewhat different software event graphs correspond to the same user event graph (see Figure 4.2). In Figure 4.2, the dashed arrows represent the mapping of user event sequences onto software event sequences. Note that the same user event sequence may correspond to more than one software event sequences due to software events that produce no user observable effect. A method is defined as functionally important with respect to the functionality of the software represented by a set of usage scenarios. This set can be divided into equivalence classes UC^+ and UC^-, respectively. This notion of a method's functional importance can be verified and further qualified by adopting ideas from mutation testing [141]. A change to the methods code body that maintains semantic and syntactic correctness of the code, i.e. a mutation, must cause change to at least one execution trace of the software. This change in execution trace is such that the trace no longer belongs to equivalence class SC^+ but SC^-.

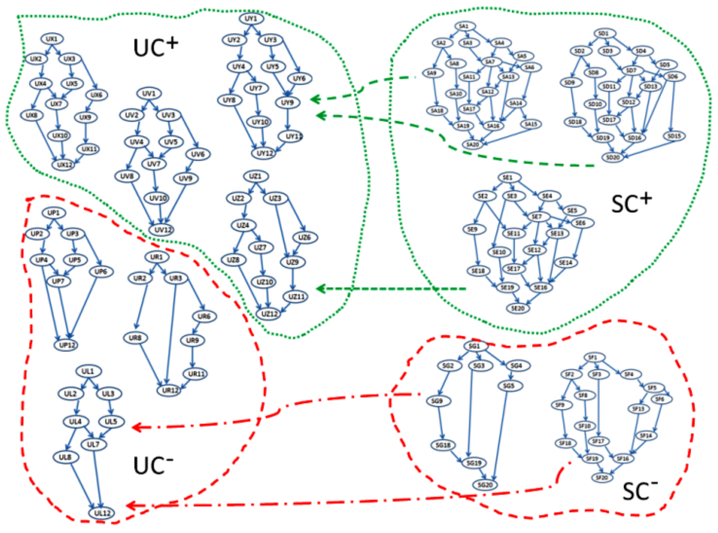


Figure 4.2 A representation of user experience (UC+, UC–) and software execution (SC+, SC–) equivalence classes.

4.2 Definition of functional importance

A method is defined to be *functionally important* if a syntactically and semantically correct change of the code of the method causes the software to produce some undesired behaviour, from the perspective of the user, with respect to a usage scenario.

4.3 Functional importance and functional requirements

The SWEBOK [132] provides the following definitions for functional and non-functional requirements of software.

"Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities."[132]

"Non-functional requirements are the ones that act to constrain the solution. Non-functional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of many other types of software requirements"[132]

Nuseibeh and Easterbrook [204], discussing the software requirements engineering, highlight the goals and constraining factors that determine software specification. The analysis of software requirements leads from goals to functional requirements, and the constraining factors provide the non-functional requirements. The functional requirements lead to design specifications where domain concepts like an "online shopping system" is mapped to a set of software functions, such as the system's necessity to have "a shopping cart" and "an online payment" subsystem amongst other things. These subsystems together compose the online shopping system. The object oriented design of the system, that involves data modelling and other considerations, would contain a set of classes that model the data and class methods that implement data exchange and transformations. One or many classes may together deliver a functional requirement. In reverse engineering concepts and feature location [66], research attempts to associate the visible system functions with the underlying source code elements that implement the functionality. The non-functional requirements determine the data and functional model, because it sets the envelope of expected behaviour, for instance "the online shop must be able to handle 100 stock items and 10 customers" and "the system must have quick response time of page transitions". In this thesis the concept of functional importance is primarily associated with functional requirements and the key method interactions that ensure delivery of the functional requirements. The verification of functional importance deals with user perception and user driven scenarios and may be mapped to non-functional requirements.

4.4 Practical verification of functional importance

The above definition allows any kind of syntactically and semantically correct changes to the code of the method. The range and kind of changes to the method's code that cause the above described effect determine the extent of functional importance of the method. Constraining the possible changes to the code allows approximation of the functional importance of methods.

A simple and rough approximation of functional importance of a method can be determined by considering the simplest generic and major alteration of the method, which is its replacement by a syntactically and semantically correct empty stub that satisfies the interface requirements of the method (i.e. input and output variables have the appropriate types and are initialized appropriately, when this is required). The method is functionally important in this approximate sense, with respect to a certain functionality of the software, if the impact of replacing the method with a satisfactory empty stub moves at least one of the software event graphs corresponding to the delivery of the functionality from SC^{+} to SC^{-}. In the following chapters this approximation of functional importance of methods is used.

4.5 Research questions

This research is based on the conjecture that when network graph representation of program data for some large object oriented software systems is analysed, the connectivity distribution pattern has small world scale free properties.

Small world scale-free graphs are known to have some very highly connected nodes, whilst the majority of the graph nodes have few connections. This leads to what is commonly referred to as fat tail distribution. If such networks are subjected to node/edge removal in a random fashion, it is seen that network fragmentation does not occur as easily if the nodes or edges removed belong to the nodes that are lightly connected. The integrity of the network is not compromised in such cases, and this has led to the claim that such networks are robust to node or edge failure. If however, the node or edge removed belongs to the highly connected minority of graph nodes, the network disintegrates and the graphs structural integrity is compromised, leading to the claim that these networks are fragile. In total, the small world scale-free networks are known to be robust, yet simultaneously fragile, to node failure.

This robust yet fragile property has been studied in various social, biological, and technological networks. It has been inferred that the network's structural integrity and system's functional integrity are highly correlated.

In the case of the internet, large studies have confirmed that router and autonomous level organisations display scale-free properties and a high degree of fault tolerance. In biological networks, the study of protein-protein interaction networks has also been found to have this property. In biological networks, many studies have used this property of highly connected nodes and characterised them as being important for the system from the functional integrity point of view.

The concept of functional importance [7, 8] is established in the context of the analysis of complex biological and social systems. For example, as stated earlier well-defined and relatively small areas of the cortex in the brain are functionally important for the generation and understanding of human speech (i.e. the Broca and the Wernicke areas of the cortex [9]). Similarly, it has been mentioned that a relatively small group of actors may play a functionally important role in maintaining successful teams of actors that can deliver highly successful movies (i.e. see the analysis of the network of movie actors [7]. In the context of software engineering, similar concepts have emerged as well. For example, [205] lists four characteristics of important software elements (e.g.

classes or methods) in the context of aspect mining: popularity (frequent usage), transitive popularity (connected by usage to popular elements), significance (connected by usage to a large number of distinct elements), and transitive significance (connected by usage to significant elements). [188] use information retrieval methods to identify methods that are important from the perspective of fixing reported bugs of complex software systems. The ability to predict such functionally important parts of a software system can facilitate efforts to improve the efficiency of software testing and program comprehension activities. In general, test prioritisation [186, 203] can also be seen as the identification of the most functionally important parts of the software onto which testing should focus.

In this research a definition of functional importance for object-oriented software systems is provided, and this forms the basis on which the following research questions arise.

Q 1. What is functional importance in context software systems?

Q 2. Can network analysis be used to discover functionally important elements of object-oriented software systems in the dynamic context?

Q 3. If network analysis can discover functionally important elements in object-oriented software systems, how can these discoveries be verified as being truly important?

A problem of large scale testing is the cost associated with the maintenance of captive testing resources. In the context of this research, we utilise the Cloud for our empirical software engineering experiment and try to answer the following question,

Q 4. Can cloud based testing be gainfully utilised for large scale software testing?

Dynamic and static analysis provides complementary views of the software system and in this research we explore the correlation between static signatures of object oriented software systems and dynamic patterns, however this is not the main research question but rather an attempt to explore potential practical applications of the techniques developed as part of this thesis.

Chapter 5. Experiment design and techniques

5.1 Introduction

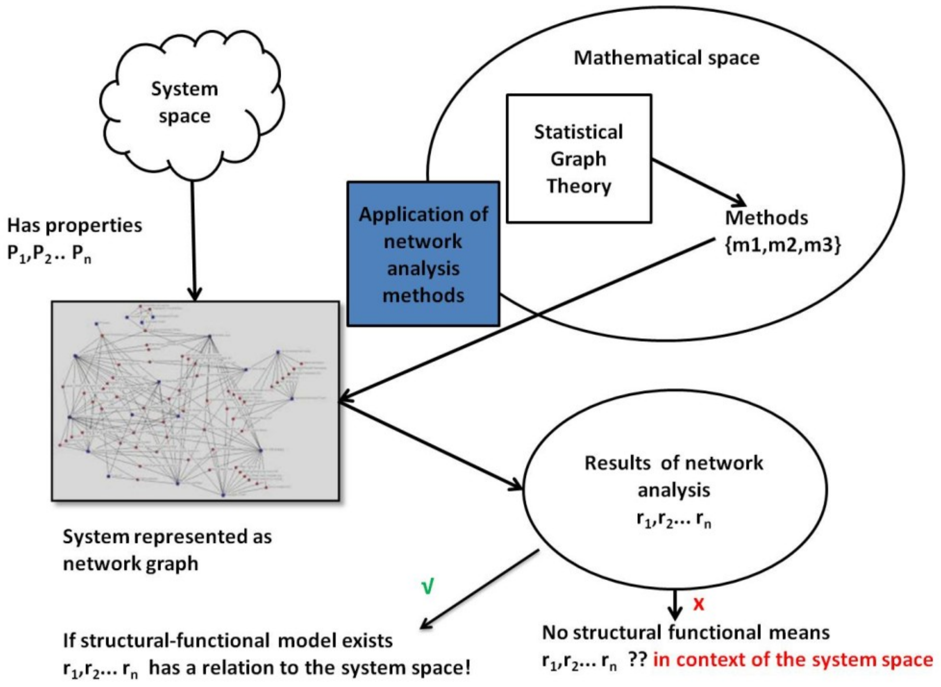
In Chapter 1, the concept of functional importance was introduced, followed by the concept and definition of functional importance in the context of object oriented software systems in Chapter 4. In this thesis, the main hypothesis is based on the concept of functional importance, and the discovery of functionally important methods in object oriented software by using complex network modelling of object interactions based on dynamic analysis of software. In this chapter, the discussion focuses on the choice of network analysis modelling, the generation of network representation from dynamic analysis data of software, and the application of network analysis techniques to predict functionally important methods and techniques used for verification of predictions. This chapter also includes a discussion of the choice of subject software, run time (dynamic) data generation, and the experiment workflow used in the empirical software engineering experiments.

In the review of background material (see Chapter 2) we have found that network graphs modelling intra-systemic interactions are used as a skeletal framework in a wide variety of fields to understand the structural integrity and functional integrity of the subject system represented by the network graph. The review also shows that network analysis of the network graph for the system enables users to study its local and global network properties, analyse the centralities of network's constituent objects based on their spatial location in the network. Analysis of the network's degree distributions facilitates reasoning about structural-functional correlations. The review also found that application of network analysis to study social [83], socio-technical [206] and biological systems[86, 87] has led to the development of empirical methods[4] used in various studies in the respective fields of development.

It was also found that network analysis has been applied in dynamic analysis based software engineering research (see Chapter 3), leading to some encouraging results [96, 106, 110, 114, 172]. The use of network analysis in software engineering has not been accompanied by development of formal or informal models to explain how the outcome of network analysis applied to software networks may be explained in terms of our understanding of software systems and the phenomena associated with object oriented software systems.

Complex networks provide a skeletal framework for data analysis, which includes a set of empirical methods that may be used to analyse structural integrity of the network. These empirical methods originate predominantly from research on social networks analysis, biological networks analysis, technological networks analysis, and data analysis in Physics. These methods are essentially similar, with certain elements that apply specifically to the field in which they were developed.

Application of network analysis methods does not depend on the system the network represents, because the analysis is based primarily on the spatial structure of nodes and edges, where an edge essentially represents a pair-wise relationship between nodes. Whether the outcome of the analysis has meaning in the context of the system is dependent on the model defined for the system (see Figure 5.1).



System space
Has properties $P_1, P_2 .. P_n$
Mathematical space
Statistical Graph Theory
Application of network analysis methods
Methods {m1,m2,m3}
System represented as network graph
Results of network analysis $r_1, r_2 ... r_n$
√
x
If structural-functional model exists $r_1, r_2 ... r_n$ has a relation to the system space!
No structural functional means $r_1, r_2 ... r_n$?? in context of the system space

Figure 5.1 Relationship between the mathematical space of statistical graph theory and the system subjected to network analysis

The concept of functional importance is the structural-functional link that relates the structurally central elements of network representations of software systems to functionally important methods in object oriented software systems.

The aim of this research is to empirically validate the predictive performance of the network analysis methods to identify structurally-functionally important nodes (objects) and edges (methods) of the object collaboration network. when used in the context of software dynamic analysis.

In the case of brain network analysis, similar structural-functional methods have been proposed [81], [9]. However, due to the constraints of data collection and clinical constraints it is not always possible to verify the functional importance of animal brain elements discovered by using network analysis [81]. In the case of large scale software systems, which are similarly complex human-engineered systems, clinical constraints to verification do not exist, and although data challenges exist they are not of a similar nature to those of clinical research.

The search for important elements in software systems has often been attempted to optimise the program comprehension process [6, 207] and software test prioritisation [186, 203]. In this context, if this research is able to verify the functional importance of method calls in object oriented software, as predicted by network analysis methods, it may contribute to two fields: software engineering and networks analysis.

5.2 Experiment design

The concept and definition of functional importance proposed in Chapter 4 is based on the scale-free network model proposed by Barabási and Bonabeau [7], and the property of simultaneous agility and fragility afforded to these networks due to the existence of highly connected network hubs. These network hubs have been found to be critical for the maintenance of structural-functional integrity of the systems represented by the network [81]. Scale-free networks are characterised by their connectivity of network nodes and edges following the Power Law distribution [102].

5.2.1 Verification of scale free power law distribution

The empirical methods used in this research to analyse the network representation of dynamic analysis data (introduced in Section 5.2.2) are applicable to data that have power law distribution [4]. Previous work on complex network modelling of OO software by Wheeldon and Counsell [114], Myers [110], and Potanin et al. [172] are all based on the assumption that network representation of dynamic analysis data for object oriented software have scale-free power law distribution $P(x) = C\,x^{-\alpha}$, with the value of exponent α between 2 and 3, i.e. $2 \leq \alpha \leq 3$.

Measuring power law distribution in natural and man-made systems is not easy [89], and this can lead to misinterpretation of data, given that there is no easy way to analytically verify power law distribution in data. The standard and widely followed technique is to use a non-parametric density indicator: the histogram of the data. The technique followed in this research is the one proposed by Newman in [102] and Clauset et al. in [91].

Verification of Power Law

The steps followed (after generation of network representation of data) are:

1. From the network representation calculate the degree of each vertex, using $d_G = d(v)\ of\ a\ vertex\ v = number\ of\ edges\ |E(v)|\ at\ v.$
2. Generation of histogram. In this experiment, histogram may be represented by

   $\ln(y) = A \ln(x) + C$ (5.1)

   Let $p(x)dx$ be a fraction of vertices with degree between $x\ and\ x + dx$. If histogram is a straight line on a log-log scale then,

   $\ln(p(x) = -\alpha \ln(x) + c$ (5.2)

   $eqn. 5.2 \xrightarrow{yields} p(x) = Cx^{-\alpha} ; C = e^{c}$ (5.3)

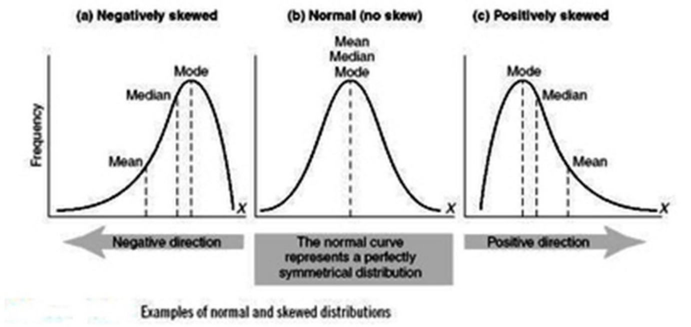
   The histogram itself can be generated using three techniques: a) log-log plot where bins are uniformly spaced; b) log-log plots with logarithmic binning; and c) cumulative distribution function. The problem with the first two is that they introduce noise into the data and also the possible loss of data due to binning scheme, making it difficult to verify if the data is really power-law distributed.

   In the context of the experiments conducted (presented in Chapters 6 and 7 of this thesis) all distributions have been verified using cumulative distribution function (CDF). In this scheme there is no loss of data because there is a value at each observed x. The computation provides us with how many values of x are at least some value $x_{min} > 0$.The CDF of power law probability distribution is also power law distributed but with an exponent $\alpha - 1$.

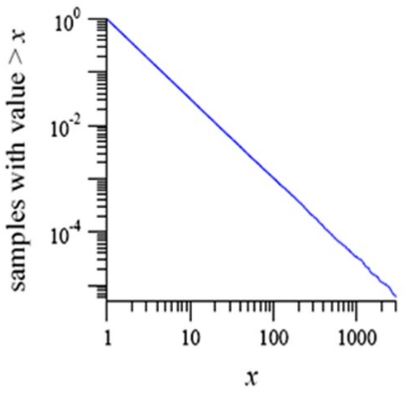
   $P(x) = \int_x^\infty p(x')dx'$ (5.4)

   $$P(x) = C \int_x^\infty x'^{-\alpha}\, dx' = \frac{C}{\alpha - 1}\, x^{-(\alpha-1)} \quad (5.5)$$

Figure 5.2 illustrates the typical shapes of histograms that are expected during statistical analysis of network connectivity distribution.. In the figure the normal distribution is placed in the centre to illustrate visual difference between histograms with positive or negative skew that is often found in the case of power law distributions. The technique typically used to verify power law distribution is visual identification of the rank - frequency plot on log-log scale gives a straight line as in equations 5.1 and 5.2.



Frequency along y axis, Connectivity along x axis



Cumulative Histogram or Rank-Frequency Plot

Figure 5.2 Skewed and normal distribution plots and Rank-Frequency Plot based on CDF gives exponent as α -1.

Value of exponent	**Statistical property**
$0 < \alpha \leq 1$	F not only has infinite variance but also infinite mean connectivity.
$1 < \alpha < 2$	F has infinite variance but finite mean connectivity.
$2 \leq \alpha \leq 3$	F has finite mean connectivity but diverging variance, standard deviation, and coefficient of variation.

Table 5.1 Summary of CDF properties based on value of exponent α.

3. The third step in this verification is to fit the histogram data, where this gives a straight line equation of the form and $y = ax + c \; and \; some \; value \; of \; R^2$, showing goodness of fit.

The value of exponent of power law α i.e. $2 \leq \alpha \leq 3$ has a mathematical significance, as highlighted by Shiner and Davison in [208] and Li et al. in [89] where the statistical properties of power law distributions are discussed in terms of stochastic and non-stochastic definitions and Complementary Cumulative distribution functions (CCDF). The following important points have been made as summarised in Table 5.1. In the stochastic context, a random variable X, or its corresponding distribution function F, is said to follow a power law or is scaling with index $\alpha > 0 \; as \; x \rightarrow \infty$, Giving $P[\, X > x \,] \;\; 1 - F(x) \approx c\, x^{-\alpha}$, for some constant $0 < c < \infty \; and \; x \rightarrow \infty$.

The point is that even with exponent of power law α i.e. $2 \leq \alpha \leq 3$, exact verification of power law distribution is not possible given the natural constraints on the size of samples that can be collected in laboratory environments [208]. In this thesis, following the discussion of Clauset et al.. in [91], the continuous variable form of CDF has been used, considering only the integral part of the values. This is purely for mathematical convenience and has no significant effect on the verification in the context of our requirements.

5.2.2 Predicting functional importance

The concept and definition of functional importance is based on the premise that the structural integrity of the network correlates to the functional integrity of the system that the network represents.

In the series of experiments performed in this work, the network generated represents objects as vertices and message interactions between objects characterised by method invocations as edges. The aim of these experiments is not to study control flow, but the importance of elements from the point of view of the functional information network and interactions within the network. The network representation derived from these experiments is undirected, but the network itself is constructed from a directed network represented by the control-flow graph or call tree of a program. In software engineering and biological networks research, especially in Brain networks, both directed and undirected network representations have been used. While some authorities [110] claim that directionality of the hierarchical network must be preserved, others [114] claim that either way no significant difference in results has been noticed. The other reason for considering undirected networks is that most of the empirical measures found in the literature that deal with the centrality of network elements are defined for undirected networks, for mathematical and computational convenience. The following section lists the network measures considered in this research.

Network analysis measures

- **Hub connection score:** this measure of an edge e is defined as the product of the connectedness values of the nodes that are connected by the edge. If the edge e connects nodes n and m, with connectedness values $v(n)\ and\ v(m)$ respectively, then

$HCS\ (e) = \ v(n).v(m)$ (5.6) .

- **Frequency weighted connection score:** this measure of an edge e is defined as the product of the connectedness values of the nodes that are connected by the edge. If the edge e connects nodes n and m, with connectedness values $v(n)\ and\ v(m)$ respectively, and the call frequency of the method corresponding to the edge e, represented by $f(e)$ or $f(e)$ is the measure of frequency of invocation of a method m represented by edges e that connect nodes n to the node representing the object instance of the class of the method m, with connectedness values v(n), this is calculated considering the call frequency of the method corresponding to the edge, $f(e|n)$, then

$WCS\ (m) = \ f(e).v(n).v(m) = HCS(e).f(e) = \ \sum_n f(e|n).v(n)$ (5.7) .

- **The total call frequency score (CFS)**: this measure of edges is the sum of the frequencies of calls of a method across all recorded calls of the method by all classes – $f(e|n)$ is the frequency of the calls of the method m represented by the edge e, originating from object instances of the class represented by node n:

$$CFS(m) = \sum_n f(e|n) \quad (5.8)$$

- **Call frequency score:** this measure is the frequency of the invocations of a method, irrespective of the caller object instances of the class (notice that this last metric was calculated using the Java Netbeans profiles)

$CFS\ (p) = \ \sum_{i=1}^{k} e_i\ \ where\ e_i\ represents\ calling\ of\ p$ (5.9)

- **Betweenness score:** this measure of a method m, represented by edges e, is the maximum betweenness score of these edges, with the edge betweenness score being the number of shortest paths connecting nodes of the network that contain the edge; there may be more than one alternative shortest paths between two nodes and the length of an edge or edge weight was set to be the inverse of the call frequency of the method, $\frac{1}{f(e_i)}$ represented by the edge:

$$BWS(m) = \max_e \left| \begin{array}{c} \{e_1, \dots, e_k\} | e \in \{e_1, \dots, e_k\}, \\ \sum_{i=1}^{k} \frac{1}{f(e_i)} \leq \sum_{j=1}^{k'} \frac{1}{f(e'_j)}, \\ \forall \{e'_1, \dots, e'_{k'}\}: \\ |nodes(e_1) \cap nodes(e'_1)| \geq 1, \\ |nodes(e_k) \cap nodes(e'_{k'})| \geq 1, \\ |nodes(e_i) \cap nodes(e_{i+1})| = 1, \\ |nodes(e'_j) \cap nodes(e'_{j+1})| = 1, \\ i = 1, \dots k;\ j = 1, \dots, k' \end{array} \right| \qquad (5.10)$$

$$\textit{where } nodes(e) \textit{ determines the two nodes connected by edge } e$$

Betweenness is the only score used in these experiments that requires calculation of shortest paths. There are various methods to compute shortest paths and the typical variants are: Single Source-Single Sink, Single Source (all destinations from a source s), and All Pairs. In this score, all pairs of shortest paths are required, which tends to be computationally intensive. In this research, a number of well-known algorithms were explored, all of which are in the tool. However, for the sake of robustness, Dijkstra's algorithm [78] was used. The running time of this algorithm varies widely, depending on the optimization techniques employed during the relaxation stage of the algorithm and on network representation. In our case, using the adjacency list and heap the running time is approximately $\boldsymbol{O(E \log V)}$. It is possible to achieve better performance by using more sophisticated schemes, which were not considered in the prototype tools.

In all the experiments on JHotdraw (discussed in Chapter 6), all of the above network analysis measures were used. The combination ranking techniques discussed in the following section were used only in experiments on JHotdraw.

In the experiments on Google Chrome (discussed in Chapter 7), all of the above network measures except Call frequency score (Eqn. 5.9) were used. Call frequency score was collected by using the Netbeans profiler in the initial validation experiments on Jhotdraw only.

Producing a ranked list of functionally important elements

Vanilla Ranking

In all cases, the calculation of scores or network metrics yields an integer number for each edge or method, which appears in the set of invoked methods extracted from the execution trace data. These lists of edges with corresponding scores, sorted in descending order of the score, provided us with a ranked list of functionally important elements, one ranked list for each of the measures calculated.

Combined ranking

In the analysis, combinations of these rankings of edges were also considered. In order to combine rankings, two methods were used. First, the sums of rankings were calculated. That is, if a method is ranked r_1^{th} according to metric M1, and r_2^{th} according to metric M2, then the combined score of the method for the combination of metrics M1 and M2 is calculated as:

$s(M1, M2) = r_1 + r_2$ (5.11).

The method was re-ranked according to this combined ranking-based score. Second, the combined score was also calculated as the product of rank values:

$s(M1, M2) = r_1 \times r_2$ (5.12).

The method was re-ranked according to the combined ranking-based scores. Combined ranking was calculated for all three scoring methods using both approaches (i.e. sums and products of rank values).

The reason for choosing this method for combination of rankings is that the scores calculated according to the different network analysis methods are not necessarily comparable (e.g. one may be a magnitude larger than the other one for all highly ranked edges). Since the distribution of the score values is likely to be not normal, normalisation and calculation of a z-score (i.e. normalised value = (value – mean)/(standard deviation)) is not meaningful, and consequently normalised scores cannot necessarily be used for the calculation of a valid combined score. This leaves us with the rank-based calculation of combined scores as the method that is sufficiently justifiable in the sense of combining comparable values, in order to avoid domination of the combined ranking by only one of the considered rankings.

5.2.3 Choice of subject software

Table 5.2 provides a tabulated list of the software chosen for empirical software engineering experiments, conducted as part of this work and a collaborative work exploring future research of network analysis software using dynamic analysis.

Serial Number	**Software**	**Version**	**Java or C++**	**Traced**	**Comments**
1	JHotdraw [209]	6.0b1	Java	Full trace	Included in this thesis and [10-12]
2	Google Chrome [210]	72930 (svn)	C++	Profile trace	Included in this thesis and [13, 14]
3	ArgoUML [211]	0.22	Java	Full trace [212]	Included in Collaborative research publication [15] and the future work section of this thesis.
4	JabRef [213]	2.6	Java	Full trace [212]	Included in Collaborative research publication [15] and the future work section of this thesis.
5	muCommander [214]	0.8.5	Java	Full trace [212]	Included in Collaborative research publication [15] and the future work section of this thesis.

Table 5.2 Table of data selected for series of experiments

In the context of the work [10-14] reported in this thesis, the choice of data JHotdraw [209] was motivated by work reported by Cornelissen et al.. in [142], and the choice of Google Chrome [210] was motivated by work reported by Voigt et al. in [143]. Jhotdraw is a medium sized application written in Java and is an exemplar implementation that incorporates design pattern, while the Chromium browser is a large scale open source application with advanced features written in C++.

In Table 5.2, data in rows with serial numbers 1 and 2 were used in all the experiments, performed to validate the concept of functional importance and for validation of network analysis measures. The comments column also lists the work's inclusion in the thesis and any publication that resulted from the experiments. Data in rows with serial numbers 3, 4 and 5 are publically available dynamic analysis data that were used with permission from SEMERU [215]. These data sets were used in a collaborative research experiment related to exploration of future research that would use both dynamic and static analysis techniques to study dynamic method stereotype distribution.

5.2.4 Collection and management of data

Collecting data program execution trace data has required significant investment of effort. This effort was primarily spent on three activities: a) preparing software build and execution environment, b) evaluating and choosing tools and techniques including deciding the format of collected data, and c) execution and recording of scenarios. There is a line of argument that supports the use of publically available dynamic analysis data instead of expending effort on data collection. The problem is that not much data, in the correct format and with the verifiable provenance, was available as open data at the start of this research in 2009. Data from SEMERU [215] became available in 2011-12, and was used with permission because most of it was found to have the necessary provenance information.

Research on software engineering that discusses empirical software engineering advocates the need to collect data in a principled manner, recording the decisions, techniques, and tools in reasonable minutiae so that it is possible to rerun the experiments and reproduce the results. This approach of recording decisions may seem cumbersome, but it helps to reduce instances of 're-invention of the wheel' [192].

Data collection for dynamic analysis usually involves some form of instrumentation of the source code or executable binary code of the software program. The following section discusses the techniques and tools evaluated and used to perform data generation. This evaluation was not done from a performance evaluation point of view, and choices are based on perception of tool performance reinforced by experiences published online and in the literature, for instance Waddington et al.. in [149].

All the software engineering experiments carried out in this research used dynamic analysis. The dynamic analysis techniques chosen were execution tracing and profiling with full tracing of control flow. The analysis in all cases was offline analysis. That is, data was first collected then analysed, as opposed to data analysis occurring simultaneously with data collection.

In software engineering execution, tracing can be performed in two ways: a) limited scenario based tracing which is analogous to grey-box testing. That is, the tester has access to the source code and is also aware of the top-level features such as menu operations, but is not aware of the underlying organisation of the source code (i.e. classes, methods and libraries) that contributes to realization of the feature. In this tracing mode, only the feature is exercised, by executing a scenario that is expected to exercise the feature and help localise the extent of its interactions with different enabling elements within the source code. This technique is similar to the technique discussed by Poshyvanyk et al.. in [188].

The second method b) is analogous to user acceptance testing of software. In user acceptance testing, the end user operates the software in a manner that mimics typical usage based on the users' expectation of the functionality of the software. In this mode, the software is exercised by using typical usage scenarios. The scenarios are not limited to just one feature but can involve many features (as in real life). In theory, this is comparable to black box testing, in which the software tester does not have access to the source code, and the primary goal of such scenarios is to simulate real life usage where one can do several tasks that can involve using a set of features. In practice, as in these series of experiments, access to the source code was available and the technique used is similar to the technique used by Cornelissen et al in [142]. In some real life situations software engineers may not have access to all of the source code, and are constrained to rely solely on execution logs or traces for reverse engineering and software maintenance activities.

Manual instrumentation

Instrumentation of source code can be attempted manually (by using print statements), and this is a common practice among software engineers attempting to debug or inspect small programs or a small part of a program. The advantage of this approach is that engineers can inject instrumentation code fragments in a pin pointed and selective manner into regions of interest, and extract data in a format that makes it easy to perform further analysis. The disadvantage is that in the context of large scale programs, with the source code spanning many thousands of lines of code and also spread across many source code files, the process is cumbersome and prone to errors.

Tool driven instrumentation

Computer assisted software engineering (CASE) tools are widely used to instrument software. These tools can be classified according to the form of the software the tool instruments take as input (i.e. source code or binary code including byte code).

Source code instrumentation

It is possible to qualify source code instrumentation further, based on whether the tool uses text parsing based on custom tokenisation and a regular expressions matching approach to insert instrument code, or some special compiler that decomposes the source and weaves in the instrumentation code.

Token based tools

In this approach, a software program or tool is used to instrument the source code or executable binary code of another program. Tool driven instrumentation can employ text analysis, tokenisation, and some custom intermediate forms to instrument un-compiled source code such as the Tau tools [216] as discussed by Geimer et al. in [150]. Tools of this type reduce the effort needed, as in the case of manual instrumentation, but some manual intervention is required when instances of code obfuscation is encountered. In cases where the instrumented code is not inspected, errors can occur in the instrumentation process, leading to collection of spurious data.

Special compiler based tools

Static analysis of source code can also be performed by using compiler dependent tools, which require manual instrumentation using special constructs or directives in the code body. Popular tools in this class are tools that utilise concepts of aspect oriented computing (AOP). In these tools, special constructs are introduced in the code body, where these constructs can be used to define rules based on patterns in the source code. When code is compiled using the special compiler, code fragments are inserted at points as and when these fragments satisfy the rules defined by the programmer. These code fragments are executed as dictated by the rules during normal program execution. There are many tools that claim to provide AOP functionality: in the case of programs written in Java, AspectJ is the most popular tool [160, 161]. Cornelissen et al. in [142] uses a custom tracing tool that uses AOP and the AspectJ tool. In this research, an AOP based tracing technique was explored at an initial stage and tracing aspect that utilised the Worm-hole pattern discussed by Laddad in [160] was used. It was found that AspectJ based tracing solutions severely affected the response performance of the application being traced. The work discussed here does not use AOP based tracing.

Static Binary instrumentation of executable systems

The class of tools similar to AspectJ, but without the need for special compilers, is based on Java2's instrumentation framework that is part of the Java Machine Tool Interface (JVMTI). An example of this tool is Eclipse Probekit [163]. In some public static, classes are written with static methods. These methods are written in standard Java and do not use any special construct. These classes and methods are inserted before or after call sites in the byte code of Java application, by utilising Java's compiler technology that supports binary instrumentation as discussed by Binder in [158]. Such techniques are often referred to as trampolining in the context of compiled language instrumentation. In this work, part of the data for [10-12] was collected using the Eclipse Probekit tool. The advantage of this tool is its relative ease of use, in comparison to the AspectJ or Tau tool suite. The disadvantage of this tool, as noticed in the most current version of Eclipse, is that it has thread safety bugs, especially in the case of large scale Java applications. Part of this problem is that IBM's Eclipse TPTP project has come to an end and no further development or maintenance is currently being carried out. In addition, this tool also introduces response performance problems in instrumented applications.

Dynamic Binary instrumentation of executable systems

Instrumentation of an un-instrumented executable software program as execution progresses, or at the start of execution, is termed dynamic binary instrumentation. At the conceptual level, most of these tools rely on some form of intermediate layer of software stack, such as virtual machines or runtime environments, to implement dynamic binary instrumentation. This is true whether we consider the case of programs written in languages like Java or C# that are compiled to bytecode, or languages like C, C++ and Fortran that are compiled using native binary. Most of these tools have been implemented in recent years with improvements to compiler technology, and are primarily aimed at communities engaged in software profiling. Profiling tools implemented the earliest form of profiling using special compiler flags, which instructed the compiler to introduce special instruction fragments to count various types of events, like frequency of function invocation or frequency of a particular path in a program being executed. Typically, profiling tools came with a set of fixed features, and did not provide users the option to implement custom profiling features without having to develop most of the tool from scratch.

Oracle's Sun Java JVMTI [217] technology provides two functionalities for binary instrumentation: a) static binary instrumentation similar to implementations like Probekit, and b) a profiling agent based framework in which a server agent acts as the runtime data collector and the subject application loads a library containing the instrumentation library which acts as data generator. The data generator and data collector execute on two separate processes, but share the Java Virtual machine’s memory and exchange data over TCP/IP or sockets. JVMTI provides the interface that can be used to write the custom profiling or tracing library. One such implementation that is widely used is Eclipse TPTP profiler and tracer[163, 182]. TPTP based tracing tools use XML4Profiling [183] to output the event trace of the application being profiled. In this work, all software in Table 4.1 has been traced using the TPTP profiling and tracing tool.

In the case of compiled languages such as C++, collection of trace data was found to be a major challenge, primarily because of a lack of quality tools, both open source and proprietary, that could be customised to produce data in a format required without the significant need to engineer a solution. The tracing tool that is used in a compiled language space is Valgrind [218].This tool represents the heavyweight approach to profiling and tracing, where it is highly customizable but also prone to errors and frequent crashes. The tool uses an intermediate format to represent binary instructions, and works by patching code with instrumentation fragments at runtime. Further details are available from Nethercote and Seward's discussion in [152].

A particularly interesting dynamic instrumentation framework that uses a virtual machine is Pin [154, 156], which is a more advanced implementation of HP's ATOM and Caliper tools [153]. Pin is similar to JVMTI, and provides users with a higher level interface to write a custom profiling and tracing tool. In this research, a custom tool was written for both Windows and Linux platforms. However the data extracted was too large, because tracing was taking place at a lower level of granularity than required. In this respect, the Intel VTune tool [219], which uses Pin based profiling, was also evaluated with the aim of using it as a tracing tool. However, its format was found to be proprietary and not customizable to our requirements.

Microsoft software development tools, predominant among which is the Microsoft Visual studio tools suite, include an enterprise class compiler for Microsoft supported languages and tools for profiling, tracing and testing. These tools belong to one of the two classes depending on the type of language. That is, interpreted languages from Microsoft like C# are called managed languages and require a runtime environment called CLR [220] similar to Oracle's JVM [221], while the other type constitutes tools meant for native executables that are compiled directly into executable PE format [222]. In this work, data was collected from Google Chrome, executable in PE format. Microsoft Visual Studio (MSVS) 2008 Team Studio and Ultimate Edition 2010 both provide access to profiling tools. These tools use a standalone profiler that is provided with an application programming interface (API) that facilitates building custom tools by using the MSVS profiler [223]. At the evaluation stage it was discovered that MS Visual studio contained a tool that could extract trace data in xml format containing caller-callee event sequences. However, it was also found that the output format was fixed and not open to easy customisation. This problem in the output data format is similar to the problem noticed in the case of Intel VTune and AQTime [224], another tool that uses the MSVS profiler. In the case of Google Chrome, data was collected using SlimTune [225], an open-source profiler that uses the MSVS profiler and a profiler meant for CLR dependent programs. The advantage of using SlimTune is that it is customisable, and in the case of this project the customisation was carried out to remove sampling of events. In most profilers, sampling cannot be removed completely, whereas in this case it was possible. The cost of removing sampling is that the tool becomes sluggish in trying to manage the large volume of events. The other advantage of this profiler is that data is stored in portable SQL-Lite database files, in a format that is easy to read and manipulate.

In addition to the tools and techniques discussed above, a range of compiler flag-based instrumentation techniques were evaluated for the purpose of extracting trace data from Google Chrome. These techniques involve the use of a) "-finstrument-functions" flag available in GnuCC [166] and b) the "/Gh" flag available in MSVC compiler as Hookit functionality [226]. A problem in using these compile time function entry and exit point instrumentations is that they can cause application crashes, and in projects like Google Chrome – with approximately 480 large projects with multiple optimizations and custom built steps – it is difficult to locate where these flags need to be introduced, because the build itself is automatically generated.

Process interference, injection and process sluggishness

In collecting execution trace data, it was necessary to manage the platform very carefully, as the tools are often unstable and susceptible to interference from other processes running on the same processor and accessing the memory system. These issues have been discussed by Ferreira et al. in [227], particularly in the context of noise injection in the data. Waddington et al. in [149] discuss various challenges in profiling and tracing multi-threaded and multi-language programs executing on multi-processor platforms, many of which we encountered during our data collection effort.

Program preparation

In order to collect dynamic analysis data, the first step is to gain access to the source code, and compile the source distribution of the software. In this context, the tools used are Netbeans, Eclipse IDEs, and Microsoft Visual Studio 2008 Team Studio edition.

It was noticed that many of the Java programs had out of date build configurations and libraries, even in projects where Maven-like tools were used. In the case of JHotdraw (version 6.01b) significant problems were encountered relating to the format of source code and name space conflicts with current Java built-in classes.

In the case of Google Chrome, the problem was with custom built program GYP, which ties in 480 projects and automatically generates platform specific optimized build scripts for the development environment being used. The optimizations often generated platform specific assembly language code, and these fail on instrumentation.

While the Java programs prepared in this project were compiled on standard desktop systems, compilation of Google Chrome required an especially large memory, multi-processor system and took 3-4 hours for each build.

Managing data

Dynamic analysis traces, from a few 100 Megabytes to 130 Gigabytes each, have been generated in this work. The total volume of raw trace data is nearly 2 Terabytes, representing 30-60 scenario runs for each software. A significant part of this data is expected to be available online.

In addition to the raw trace data, each processed data file, and various files that contain data from the analysis, and results have also been saved. These files have been tagged and named in a unique manner so that it is possible to go from results to the source data. In total, the data – comprising source code, built binaries, instrumented binaries, raw traces and analysis files – for the two systems included in this thesis is nearly 600 Gigabytes.

Programs have mostly been stored on a subversion server, and data has been stored on the Amazon Cloud S3 storage service and on SAN disks available at Newcastle University.

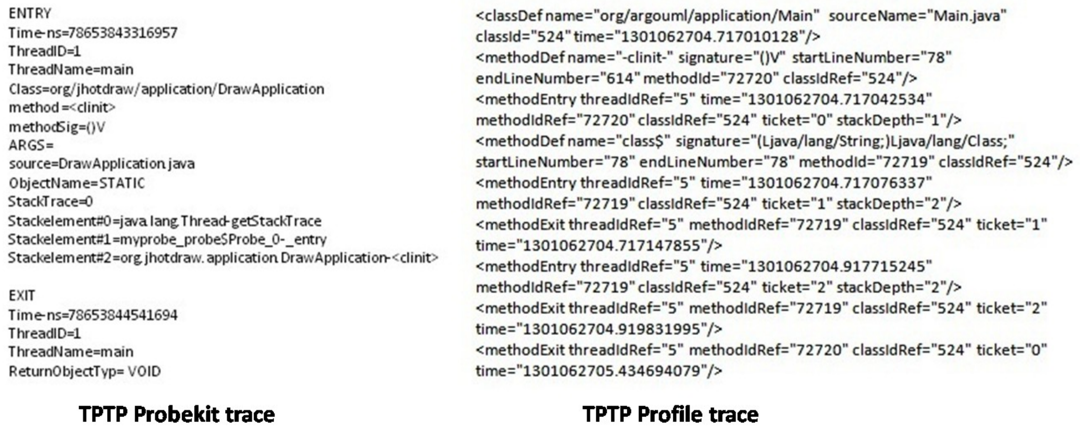
5.2.5 Data analysis using Network Analysis

Generation of network representation

Generation of network representation, starting with raw trace data, is a five step process. This process was followed and implemented in the tool chain developed for this research.

Step 1:

In this work, raw trace data was generated in three formats: a) Probekit trace in ASCII text using custom static instrumentation of Java bytecode (see Figure 5.3); b) TPTP profile trace in XML4Profiling [183] ".trcxml " format (see Figure 5.3); and c) in SQL Lite data base format (see Figure 5.4).



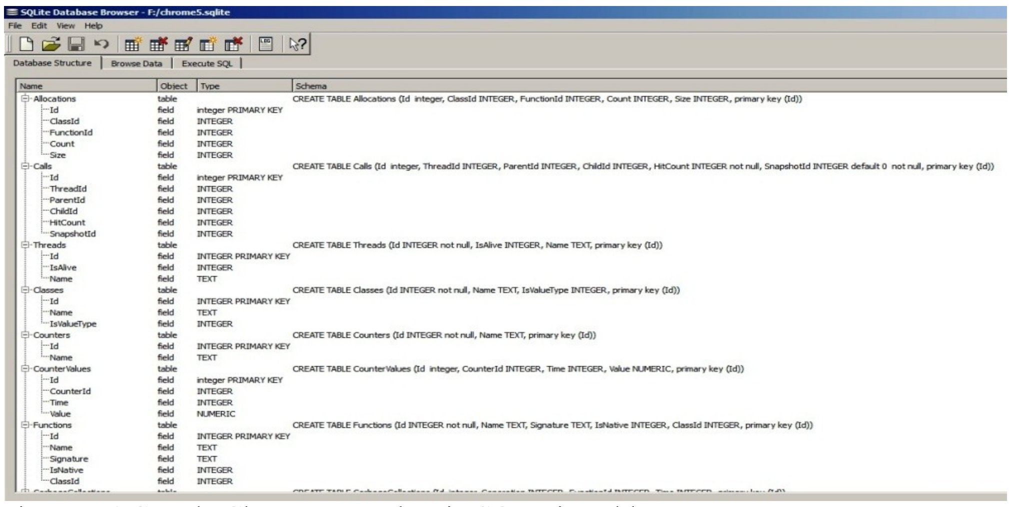
```
ENTRY
Time-ns=78653843316957
ThreadID=1
ThreadName=main
Class=org/jhotdraw/application/DrawApplication
method =<clinit>
methodSig=()V
ARGS=
source=DrawApplication.java
ObjectName=STATIC
StackTrace=0
Stackelement#0=java.lang.Thread-getStackTrace
Stackelement#1=myprobe_probe$Probe_0-_entry
Stackelement#2=org.jhotdraw.application.DrawApplication-<clinit>

EXIT
Time-ns=78653844541694
ThreadID=1
ThreadName=main
ReturnObjectTyp= VOID
```

TPTP Probekit trace

```
<classDef name="org/argouml/application/Main"  sourceName="Main.java"
classId="524" time="1301062704.717010128"/>
<methodDef name="-clinit-" signature="()V" startLineNumber="78"
endLineNumber="614" methodId="72720" classIdRef="524"/>
<methodEntry threadIdRef="5" time="1301062704.717042534"
methodIdRef="72720" classIdRef="524" ticket="0" stackDepth="1"/>
<methodDef name="class$" signature="(Ljava/lang/String;)Ljava/lang/Class;"
startLineNumber="78" endLineNumber="78" methodId="72719" classIdRef="524"/>
<methodEntry threadIdRef="5" time="1301062704.717076337"
methodIdRef="72719" classIdRef="524" ticket="1" stackDepth="2"/>
<methodExit threadIdRef="5" methodIdRef="72719" classIdRef="524" ticket="1"
time="1301062704.717147855"/>
<methodEntry threadIdRef="5" time="1301062704.917715245"
methodIdRef="72719" classIdRef="524" ticket="2" stackDepth="2"/>
<methodExit threadIdRef="5" methodIdRef="72719" classIdRef="524" ticket="2"
time="1301062704.919831995"/>
<methodExit threadIdRef="5" methodIdRef="72720" classIdRef="524" ticket="0"
time="1301062705.434694079"/>
```

TPTP Profile trace

Figure 5.3 Trace fragments of TPTP Probekit trace and TPTP Profile trace.



Name	Object	Type	Schema
Allocations	table		CREATE TABLE Allocations (Id integer, ClassId INTEGER, FunctionId INTEGER, Count INTEGER, Size INTEGER, primary key (Id))
Id	field	integer PRIMARY KEY	
ClassId	field	INTEGER	
FunctionId	field	INTEGER	
Count	field	INTEGER	
Size	field	INTEGER	
Calls	table		CREATE TABLE Calls (Id integer, ThreadId INTEGER, ParentId INTEGER, ChildId INTEGER, HitCount INTEGER not null, SnapshotId INTEGER default 0 not null, primary key (Id))
Id	field	integer PRIMARY KEY	
ThreadId	field	INTEGER	
ParentId	field	INTEGER	
ChildId	field	INTEGER	
HitCount	field	INTEGER	
SnapshotId	field	INTEGER	
Threads	table		CREATE TABLE Threads (Id INTEGER not null, IsAlive INTEGER, Name TEXT, primary key (Id))
Id	field	INTEGER PRIMARY KEY	
IsAlive	field	INTEGER	
Name	field	TEXT	
Classes	table		CREATE TABLE Classes (Id INTEGER not null, Name TEXT, IsValueType INTEGER, primary key (Id))
Id	field	INTEGER PRIMARY KEY	
Name	field	TEXT	
IsValueType	field	INTEGER	
Counters	table		CREATE TABLE Counters (Id INTEGER not null, Name TEXT, primary key (Id))
Id	field	INTEGER PRIMARY KEY	
Name	field	TEXT	
CounterValues	table		CREATE TABLE CounterValues (Id integer, CounterId INTEGER, Time INTEGER, Value NUMERIC, primary key (Id))
Id	field	integer PRIMARY KEY	
CounterId	field	INTEGER	
Time	field	INTEGER	
Value	field	NUMERIC	
Functions	table		CREATE TABLE Functions (Id INTEGER not null, Name TEXT, Signature TEXT, IsNative INTEGER, ClassId INTEGER, primary key (Id))
Id	field	INTEGER PRIMARY KEY	
Name	field	TEXT	
Signature	field	TEXT	
IsNative	field	INTEGER	
ClassId	field	INTEGER	

Figure 5.4 Google Chrome trace data in SQL Lite table.

Step 2:

Raw data is processed into call sequence data using the form given below:

<call sequence serialized according to stack depth > Caller class: caller method # signature: callee class:callee method # signature # threadid #<ENTRY/EXIT>

Step 3:

The data is then transformed into a modified form of network data in Rigi standard format [228], as discussed in [142]. The use of this format is not strictly necessary. However, this format was used in initial exploratory work on reproducing the results of Cornelissen et al. [142]. The network analysis tools developed in this research required input data in Rigi format files.

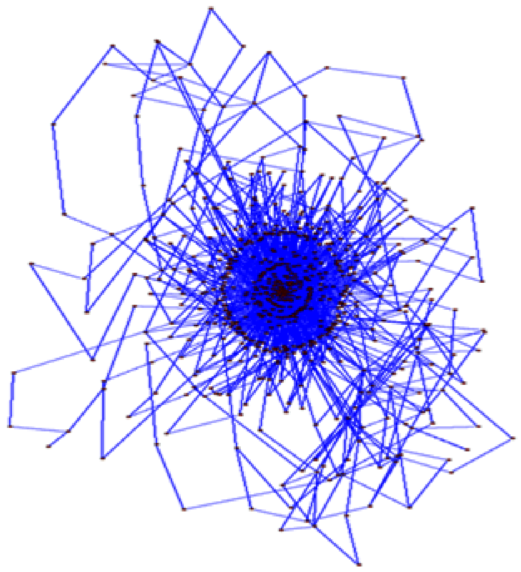


Figure 5.5 Google Chrome object-method calls network.

Step 4:

Data in Rigi format was transformed into one of the standard network visualisation formats. In this thesis, all visualisation was generated using Pajek [229]. Other visualisation tools and formats such as Graphviz [230] and JUNG [231] were also evaluated, and tools produced as part of this work can produce visualisation in all the three formats. Figure 5.5 is a network graph visualisation generated from Google Chrome data.

Step 5:

The next step of data transformation that occurs within the tools is generation of the adjacency matrix and incidence matrix of the data. In this analysis, all network representation was generated using incidence matrix representation. Incidence matrix representation is a more compact representation and is particularly suited for large sparse networks.

Let G be a graph with n vertices, m edges, and no self-loop. The incidence matrix A of G is $n \times m$ matrix $= [\, a_{ij}]$, whose n rows correspond to the n vertices and the m columns correspond to m edges such that:

$$a_{ij} = \begin{cases} 1, if\ edge\ m_j\ is\ incident\ on\ i^{th}\ vertex \\ \quad 0, otherwise \end{cases}$$

In this thesis, all incidence matrices ignored self-loops in the non-weighted networks. Ignoring self-loops does not have any effect because the shortest path is zero in such cases. In weighted networks, the shortest path has a value determined by the weight associated with the self-loop. In these cases, the effect of self-loops was considered, but with weights taken as the inverse of frequency of invocation, the contribution was small. In the case of connectivity distribution, self-loops make no difference. Incidence graphs provide the most memory efficient data structure to store network data. However, a current drawback is the representation of concurrency, in which multiple incidence graphs are used to represent different threads of execution. This particular technique is inefficient.

5.2.6 Verification of network analysis results

The network representation, generated from scenario based execution tracing of a software application in our research, consists of a set of vertices that represent objects or classes and edges that represent method invocations. The assumption is that during scenario executions, the user did not encounter any user observable deviations from expected normal operation of the software. That is, the software functioned normally. Therefore, the network is considered structurally sound from the point of view of functional integrity.

The verification is based on the hypothesis that if the network is perturbed at the level of edges, it is likely to lead to one of the following outcomes: a) the software will experience a fatal crash; b) the software will experience errors but not crash (i.e. it will experience run-time state corruption); and c) the software will continue to function normally.

In scale-free power law distributed small-world network parlance, network analysis is supposed to predict the edge elements of the system that are central for structural, and consequently functional, integrity. The approach used in this work to verify this is to introduce a perturbation in the edge interaction network. If the network elements perturbed exhibit low connectivity or a tail of distribution elements, then the system is likely to continue to operate normally. However, if the elements belong to the few highly connected hubs it is likely to lead to network fragmentation of some sort, such that it is functionally impaired.

Generation of mutant programs

In software engineering, an often employed technique is the introduction of dummy code that is syntactically and semantically correct. That is, code that accepts inputs and returns outputs such that type errors do not occur. One can also view dummy code introduction as mutant generation, and in our case the mutant is specification complaint and the application compiles as normal.

The replacement of a method with an empty stub with an appropriate interface is a maximal alteration of the method. Other kinds of interface that contain compatible and syntactically and semantically correct alterations of the method are also possible. If a method is functionally important for the delivery of functionality of the software, the maximal alteration of the method that is applied will have a significant impact on the delivery of functionality of the software. However, such maximal alteration may also lead to significant alteration of the delivery of the software functionality of other methods as well, that have much less impact on the delivery of the software functionality if their alteration is less extensive. Thus, in principle it is an interesting question to know the minimum level of alteration of a method that causes significant impact on the delivery of functionality of the software. In particular, it would be useful to have a principled method of imposing varying levels of alterations of methods, in order to make possible the investigation of the extent of functional importance of methods.

It is often found that OO software contains empty methods, where in such cases maximal alteration makes no difference because in effect mutation will have no effect on the method body.

Verification of functional importance

The verification process aims to execute mutant builds of the software, following the scenarios executed at the time of data collection. In this regard the ranked list is marked 1 to indicate fatal failure and state corruption, and 0 if the software keeps functioning normally with minimal or no user-observable effect due to the underlying mutation. The result of analysis is discussed in chapters 6, 7 and 8.

Producing random lists to establish a baseline for comparison

The premise of these experiments is that network analysis can identify functionally important elements. The question that naturally occurs is: with what degree of accuracy is it possible to determine when a method or a set of methods is chosen at random that some or all of the methods are functionally important? In the experiments discussed here, 100 methods were chosen at random from a list of methods that appeared in the traces. These methods were evaluated by mutation analysis. The result of the mutation test on the random list of methods provided a baseline against which prediction performance was compared.

5.2.7 Experiment workflow

In this research all experiments followed the stages that appear in the schematic illustration in Figure 5.6, starting with the source code on the left, generation of instrumented executable, trace data collection from scenario executions, data analysis of data, evaluation, and verification of data analysis results.

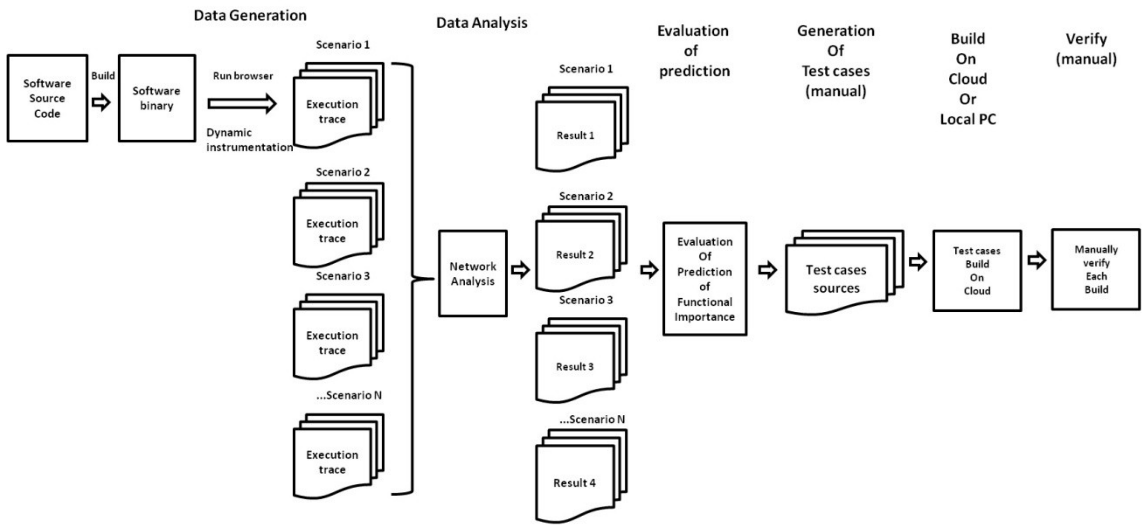


Figure 5.6 Diagrammatic illustration of experiment workflow.

The elements of this workflow include the choice and preparation of the instrumented software, scenario driven execution trace data collection, data analysis using network analysis methods, the process of evaluation of data analysis, and verification of the results. Post verification is the interpretation of the results in the context of relevant aims and understanding of software engineering of object oriented systems. Post verification results and interpretation are discussed in chapters 6 and 7.

5. 3 Discussion

In the preceding sections of this chapter, the discussion focused on the design of experiments to perform verification of the concept of functional importance, and the network metrics that can be used to predict functionally important methods or objects.

Subsequently, the workflow diagram identified the different stages of our experimental process, consisting of program preparation, data collection, network representation generation and analysis, and showed how these stages are connected, culminating in verification of the concept of functional importance. In the context of this research, use of the Cloud has been discussed in terms of scalable software testing technique and as an enabling resource. In the remaining chapters of this thesis, the focus moves from discussion of experimental techniques to the empirical software engineering experiments and their results.

Chapter 6. Experiments on JHotdraw

6.1. Introduction

In Chapter 5, Figure 5.6 gives the schematic diagram of the experimental process workflow used in all empirical software engineering experiments presented in this thesis. The discussion in Chapter 5 focused on experiment design, data collection, data verification, network analysis, and finally the process of verification of functional importance. In this chapter, the outcome of data analysis will be discussed. Mechanistic process details concerning data collection and the generation of network representation have not been repeated here, except for a brief description of the typical usage scenarios used to generate the execution traces.

In experiments where analysis of feature location [67] is the aim, scenarios often involve executing the program using a short scenario [188] that exercises specific functionalities, like open and save file by using the menu option for this feature available through the user interface. Short scenarios generate full traces of smaller size, thereby making the process of mining the data to link feature to code relatively easier than the case where a typical usage scenario is used, which leads to large traces that contain events from many features. In this research the aim is to explore whether functionally important method interactions can be discovered using network analysis, based on typical usage scenarios of an application. A typical usage scenario is simulated by executing scenarios similar to the scenarios used in user acceptance testing of software.

The JHotdraw software [232] is an object oriented framework for graphical applications written in Java. Although the 2D graphics framework was originally written (by Gamma and Eggenschwiler) as an exemplar to demonstrate the use of object oriented design patterns [70], it has several features that were unique to it at the time of its development, such as the use of generic containers and advanced features for graphical user interface development like a desktop pane. These features have been incorporated as standard features in Java's second edition, specifically, Java Swing and AWT classes. The standard distribution of JHotdraw [232] contains many example applications, one of which is the desktop application for drawing and manipulating geometric shapes and image files. One may find that JHotdraw is comparable to the popular application 'Paint' available on Windows platforms.

The version of JHotdraw [232] used in empirical experiments discussed in this thesis is 6.0b1. This version has approximately 71267 lines of code (LOC) including test packages, organised into 16 packages, and containing 344 classes. The framework has 188 classes including 20 core framework classes. These classes implement approximately 3694 methods including test classes. Figure 6.1 gives a class hierarchy diagram of the core Jhotdraw framework, and Figure 6.2 gives the UML diagram that presents the organisation of core framework classes which the exemplar application used in this experiment relies on.

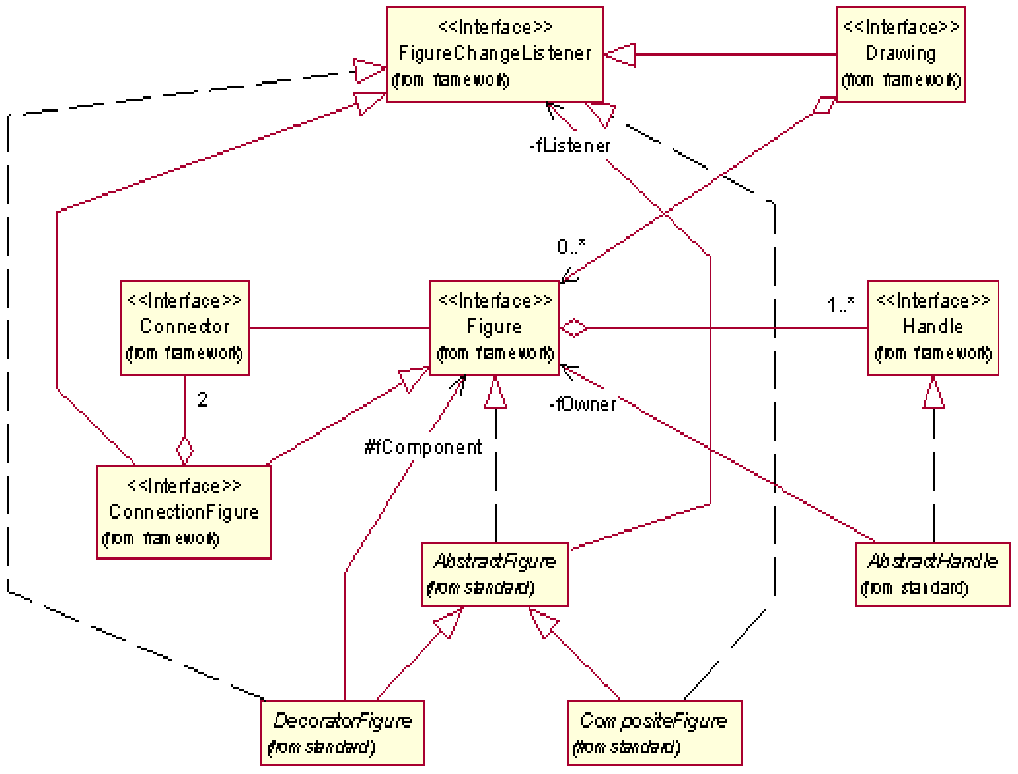


Figure 6.1 Class hierarchy diagram of JHotdraw.

In Figure 6.2, the draw application class is highlighted in red because that is the main class of the Jhotdraw framework that needs to be extended by any 2D graphics application built using the JHotdraw framework. The sample standalone graphics application uses approximately 165 framework classes and implements 1000 methods, including constructors and utility methods.

The trace data collected using Eclipse TPTP framework, when spread across all scenarios, achieved 60% coverage of the framework classes' methods and 100 % coverage of the application consisting of 1 main class.

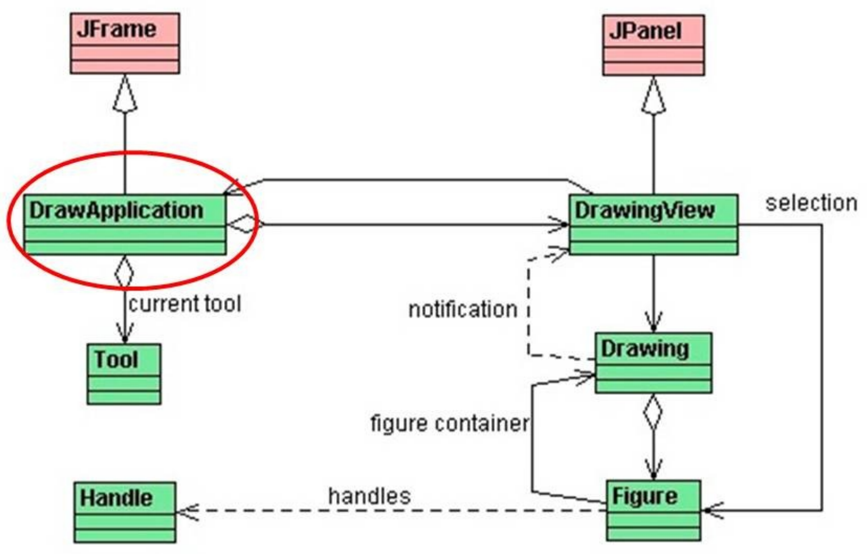


Figure 6.2 The UML diagram of core JHotdraw framework.

The scenarios used in this experiment to generate execution traces were first briefly discussed by Cornelissen et al. in [142] . In these scenarios, a new drawing pane is opened from the menu option, then five standard geometric figures: a triangle, square, rectangle, circle and a rhombus are selected from the shapes menu to be drawn in the drawing pane. The scenarios used in this project extend these scenarios: by creating three panes; using copy, cut and paste functions; group and ungroup functions; setting image layers; mixing images and standard figures; figure animation in single and multiple panes; and changing figure attributes like colour, line and border pixel width.

In the initial exploratory experiments where the technique was still being developed, recording of the scenario was non-textual. These scenarios were mostly saved as a series of final state screen shots. A problem with this approach is that a significant amount of information necessary to rerun the scenarios is lost, and this is the case even with [142]. This is not ideal and considerable effort was made to find tools that could record scenarios by recording mouse clicks, and rerun these recordings. It was found that tools that record mouse clicks are unreliable. The other option used to record scenarios was to use video screen capture software, which allows annotations to be made. The problem with video screen capture is that such software interferes with the tracing application, adversely affecting an already slow process. In later traces the technique used by the feature location community, of textually describing scenarios, has been adopted, drawing inspiration from the data made available on the SEMERU website [215]. The problem of exact replication still remains, because users often interpret what was meant in the textual description in slightly different ways, and as a result, with existing tools, it seems exact retracing of steps may not be possible.

6.2 Data from execution trace

Figure 6.3 is a composite of two figures: Figure (6.3 A) is the network representation of JHotdraw 6.0b1, where nodes represent objects and edges represent method calls; and Figure (6.3 B) is a graph of connectedness distribution. The distribution of the log (connectedness) values of the nodes of the network links together with the best linear fit line and its equation, and the R^2 value is given in the upper right corner (R^2 value close to 1 indicates a good fit between the data points and the linear relationship estimation).

In the execution traces, 165 classes of JHotdraw 6.0b1 were found to have been covered by the execution scenarios. The network representation contains nodes that represent the object of these classes and edges represent method calls.

The connectedness distribution of the network nodes refers to the connectedness of a node as the number of edges connecting a node to other nodes. Considering all edges for all nodes representing classes, the best fitting distribution of the connectedness values is log-linear (see Figure 6.3B).

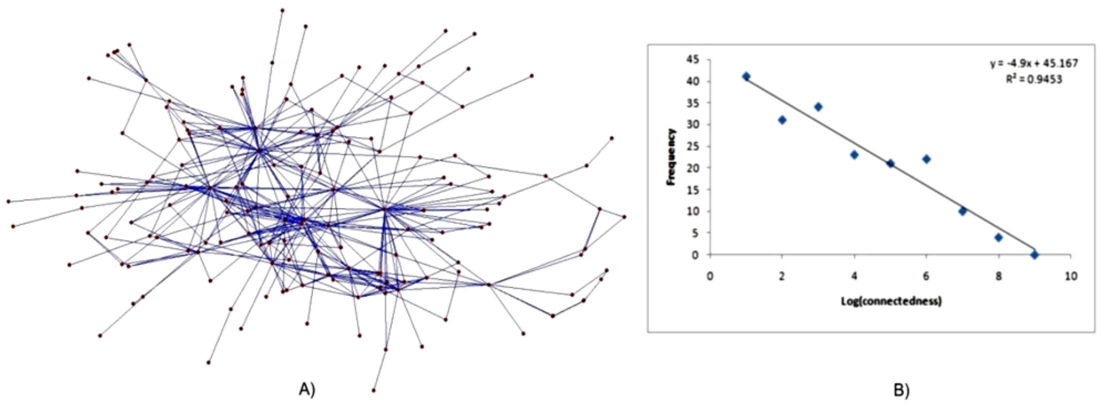


Figure 6.3 The network representation of the JHotdraw message interaction network.

The probability density function of the connectedness values is found to be

$$p(x = a) = \frac{-4.9 \ln(a) + 45.167}{a} \quad (6.1)$$

This does not represent probability density function as corresponding to scale free power law e.g. $p(x) = Cx^{-\alpha}$; $C = e^{c}$, where the literature on real world systems claims the value of α should be in the range $2 \leq \alpha \leq 4$. However, in the distribution presented here the tail is much longer than an exponential distribution, $p(x) = Ce^{-\alpha}$, which leads us to conclude that the data may be appropriate for network analysis.

6.3 Network Analysis of data

6.3.1 Vanilla analysis of network measures

The network analysis measures (introduced in Chapter 5) used in the data analysis are presented below for convenience. In Table 6.1, the top-5 methods of JHotdraw 6.0b1 in the top-20 ranked list is presented. The ranking was produced by network analysis of the following scores: HCS, WCS, and BWS. Table 6.1 also includes the top-5 methods that appear in the list of methods from the Netbeans profiler ranked by frequency of call, presented as the CFS(p) score. The measures used to generate these rankings were discussed earlier in Chapter 5:

- Hub connection score: $HCS\ (e) = v(n).v(m)$ (5.6)
- Weighted connection score:

$$WCS\ (m) = f(e).v(n).v(m) = HCS(e).f(e) = \sum_{n} f(e|n).v(n) \qquad (5.7)$$

- Netbeans call frequency:

$$CFS\ (p) = \sum_{i=1}^{k} e_i\ ; where\ e_i\ represents\ calling\ of\ p \quad (5.9)$$

- Betweenness score:

$$BWS(m) = \max_e \left| \begin{array}{c} \{e_1, \dots, e_k\} | e \in \{e_1, \dots, e_k\}, \\ \sum_{i=1}^{k} \frac{1}{f(e_i)} \le \sum_{j=1}^{k'} \frac{1}{f(e'_j)}, \\ \forall \{e'_1, \dots, e'_{k'}\}: \\ |nodes(e_1) \cap nodes(e'_1)| \ge 1, \\ |nodes(e_k) \cap nodes(e'_{k'})| \ge 1, \\ |nodes(e_i) \cap nodes(e_{i+1})| = 1, \\ |nodes(e'_j) \cap nodes(e'_{j+1})| = 1, \\ i = 1, \dots k;\ j = 1, \dots, k' \end{array} \right| \qquad (5.10)$$

$$where\ nodes(e)\ determines\ the\ two\ nodes\ connected\ by\ edge\ e$$

All of these scores, except CFS (Equation 5.9), were generated using the Netbeans profiler. In this context, it should be mentioned that both Eclipse Probekit and TPTP Agent profiler can be used to generate call frequency information. In the early phase of these experiments, the Probekit based instrument was found to be unreliable in gathering invocation frequency information, resulting in the use of Netbeans profiler [233]. In the course of this research, it was realised that invocation frequency information can be extracted from the full trace too, simply by counting trace events with identical end points and edges (i.e. methods). This is in fact a relatively more efficient way of generating invocation frequency information, because it excludes the possibility of sampling induced errors, if one assumes that full traces capture all events. In most of the scores generated, this technique has been used.

Betweenness is the only score used in these experiments that requires the calculation of shortest paths. There are various ways to compute shortest paths. The typical algorithmic variants for computing the shortest path between two nodes in a network are Single Source-Single Sink, Single Source (all destination nodes from a source node s), and All Pairs of nodes. In the BWS score, all pairs of shortest paths are required, which tends to be computationally intensive. In this research, a number of well-known algorithms were explored and implemented using the prototype network analysis tool. However, for the sake of robustness, Dijkstra's algorithm [78] was used, where the running time of shortest path computation using Dijkstra's algorithm can vary depending on the optimization techniques employed at the relaxation stage of the algorithm, and reliant on the network representation data structure on which the algorithm is applied. In our case, using the adjacency list and the heap, the running time is approximately $O(E \log V)$ where E is the number of edges and V is the number of vertices/nodes. Using Dijkstra's algorithm without optimization can lead to the worst case performance of $O(|E| + |V| \log |V|)$. It is possible to achieve better performance by using more sophisticated schemes, which were not considered in the prototype tool.

In Section 5.2.4, the discussion focused on the verification of predictions of functional importance, produced by each of the network analysis measures considered. Table 6.1 presents a summary of the mutation analysis, performed to validate the predictions produced by network analysis score based ranking. The column labelled ‘name/results’ contains both the name of the scene and the results of the network analysis. For instance, scn1_WCS implies scenario 1 and validation of the ranking produced by WCS scoring. The column labelled ‘Fatal’ captures mutation testing which led to complete failure or crash of the Jhotdraw software. Similarly, ‘Non-Fatal’ captures non crash failures, ‘No effect apparent’ means no effect of the mutation was noticed during testing, and ‘Non JHotdraw’ indicates the presence of a method on the list of predictions that did not belong to the JHotdraw software. Table 6.1 lists the summary of validation testing of the top 20 predictions, generated from 4 scenarios by each of the network measures considered. This table does not include the testing based on Netbean’s profiler based CFS(p) measure. In all cases, the top 50 predictions were analysed. However, for brevity only the top 20 are presented in the thesis. The results across all scenarios are identical even though the extraneous scenarios were not used. Table 6.2 provides the statistical analysis performed after validation mutation testing was complete. In this case, the figures in the table are from scenario 4.

name/results	Fatal(2)	Non Fatal(1)	No Effect Appatent	Non JHotdraw
scn1_WCS	6	6	8	0
scn1_HCS	7	6	7	0
scn1_BWS	4	5	11	0
scn2_WCS	6	6	8	0
scn2_HCS	8	6	6	0
scn2_BWS	3	5	12	0
scn3_WCS	6	6	8	0
scn3_HCS	8	5	7	0
scn3_BWS	3	7	10	0
scn4_WCS	12	1	7	0
scn4_HCS	13	0	7	0
scn4_BWS	10	0	10	0

Table 6.1 Tabulation of results of verification of predictions of functional importance of JHotdraw.

		cml_bws	cml_bws/rank		cml_wcs	cml_wcs/rank		cml_hcs	cml_hcs/rank	cml_cfs/rank				
Rank	bws	0		wcs	0		hcs	0			Random	RUL	RLL	R-Avg
1	0	0	0	1	1	1	1	1	1	0	0	0.6	0.49	0.549037
2	0	0	0	1	2	1	1	2	1	0.5	0	0.6	0.49	0.549037
3	0	0	0	1	3	1	1	3	1	0.666666667	0	0.6	0.49	0.549037
4	0	0	0	0	3	0.75	1	4	1	0.75	0.25	0.6	0.49	0.549037
5	0	0	0	1	4	0.8	0	4	0.8	0.8	0.4	0.6	0.49	0.549037
6	1	1	0.166666667	1	5	0.833333333	1	5	0.833333333	0.833333333	0.5	0.6	0.49	0.549037
7	0	1	0.142857143	1	6	0.857142857	1	6	0.857142857	0.857142857	0.571429	0.6	0.49	0.549037
8	1	2	0.25	0	6	0.75	0	6	0.75	0.75	0.625	0.6	0.49	0.549037
9	0	2	0.222222222	1	7	0.777777778	1	7	0.777777778	0.777777778	0.666667	0.6	0.49	0.549037
10	1	3	0.3	0	7	0.7	1	8	0.8	0.8	0.6	0.6	0.49	0.549037
11	0	3	0.272727273	1	8	0.727272727	0	8	0.727272727	0.727272727	0.636364	0.6	0.49	0.549037
12	1	4	0.333333333	0	8	0.666666667	1	9	0.75	0.666666667	0.583333	0.6	0.49	0.549037
13	1	5	0.384615385	0	8	0.615384615	1	10	0.769230769	0.615384615	0.538462	0.6	0.49	0.549037
14	1	6	0.428571429	1	9	0.642857143	0	10	0.714285714	0.571428571	0.5	0.6	0.49	0.549037
15	1	7	0.466666667	1	10	0.666666667	0	10	0.666666667	0.533333333	0.466667	0.6	0.49	0.549037
16	0	7	0.4375	1	11	0.6875	0	10	0.625	0.5625	0.5	0.6	0.49	0.549037
17	1	8	0.470588235	1	12	0.705882353	1	11	0.647058824	0.588235294	0.529412	0.6	0.49	0.549037
18	1	9	0.5	0	12	0.666666667	0	11	0.611111111	0.611111111	0.555556	0.6	0.49	0.549037
19	0	9	0.473684211	0	12	0.631578947	1	12	0.631578947	0.631578947	0.578947	0.6	0.49	0.549037
20	1	10	0.5	0	12	0.6	1	13	0.65	0.65	0.55	0.6	0.49	0.549037

Table 6.2 Results verification of the Top 20 ranked methods of JHotdraw.

Table 6.2 illustrates how the results of validation testing led to calculation of the probability of success for the predictions. In this table, fatal and non-fatal failures are scored as 1, and where there was no effect apparent this is marked as 0. The cumulative value of score is generated per rank per measure, denoted by the column prefixed 'cml', and this is followed by calculation of cumulative score of density per rank position for each of the considered measures. This value is stored in the column labelled 'cml_measure/rnk' (e.g.cml_HCS/rnk).

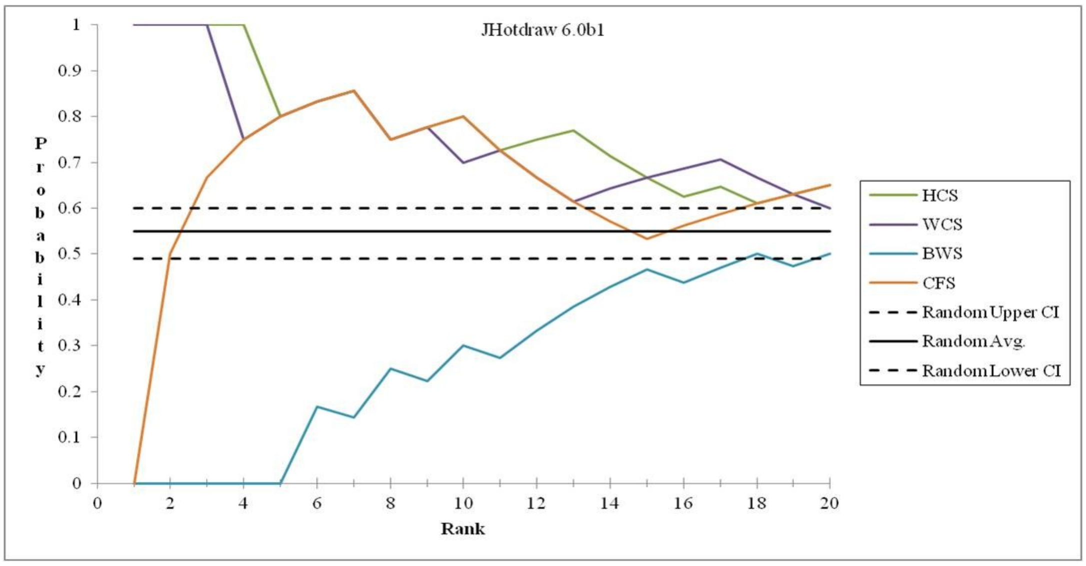


Figure 6.4 The performance of vanilla ranking produced by edge ranking methods.

	HCS scoring		**WCS scoring**
Rank	**Method**	**Rank**	**Method**
1	standard.StandardDrawingView.tool	1	util.PaletteButton.mousePressed
2	standard.StandardDrawingView.paintComponent	2	contrib.zoom.ZoomDrawingView$3.Constructor
3	contrib.AutoscrollHelper.Constructor	3	contrib.zoom.ZoomDrawingView$2.mouseMoved
4	framework.FigureAttributeConstant.getName	4	application.DrawApplication.toolDone
5	framework.FigureAttributeConstant.hashCode	5	contrib.AutoscrollHelper.Constructor
	BWS scoring		**NetBeans call frequency**
Rank	**Method**	**Rank**	**Method**
1	standard.StandardDrawingView.tool	1	framework.FigureAttributeConstant.hashCode
2	framework.FigureAttributeConstant.getName	2	figures.FigureAttributes.get
3	standard.StandardDrawingView.paintComponent	3	figures.AttributeFigure.getAttribute
4	framework.FigureAttributeConstant.hashCode	4	figures.AttributeFigure.getDefaultAttribute
5	framework.FigureAttributeConstant.setID	5	framework.FigureAttributeConstant.equals

Table 6.3 The top-5 ranked methods of JHotdraw 6.0b1.

Discussion

In Figure 6.3 A and Figure 6.3 B presented above, the network representation and the graph of the connectedness distribution extracted from network graph representation of execution traces are shown. In these experiments it was found that JHotdraw network graphs tend to fit log-linear or log-normal connectivity distribution. This may be explained by considering two facts. First, JHotdraw is a medium size software, and only 165 classes were covered by the extensive scenarios used in these experiments. Second, the exemplar application built on top of the framework uses a small number of key GUI classes that act as aggregators. The first point leads to the notion of constraints to network growth by design, whilst the second point leads to the notion of the emergence of giant components which have a large number of incoming links. If the classes 'StandardDrawing', 'DrawingEditor' and 'MDI_DrawApplication' are considered, it is seen that the DrawingEditor interface has been designed to act as a mediator, and MDI_DrawApplication is a concrete implementation of the mediator design pattern. All other classes, such as Tools, are made available only through active figure objects that are drawn in the drawing editor. If fact this points to small-world single scale networks discussed by Amaral et al. [85], where such networks are not scale-free, but borderline cases that are constrained by design. It is quite possible that a different application developed using the JHotdraw framework would display different connectivity properties. In this case, the results indicate a hub-like structure may exist, making it a good candidate for application of network analysis.

Table 6.3 presents the result of network analysis without any combination of scores (i.e. the vanilla analysis). It is seen from this table that StandardDrawing appears as the top class in score for HCS and Betweenness (BWS) (i.e. when it is on the shortest path). It is logical to expect this, because this object acts like a container, holding other objects and associated tools that operate on the figures. In the list of invocation frequency extracted from Netbean's profile invocation frequency, it can be seen that the methods associated with the object of the Figures class appear in the top level on the list. This is also logical because invocation frequency for figure-related methods is likely to be much higher than on the window that contains the methods. A figure is likely to invoke methods more frequently to keep redrawing itself as a response to a user's mouse clicks. These invocations are not normally expected to have influence on the container such as a panel within which the figure object is drawn. Chapter 5 introduced the steps involved in network generation, verification and analysis. In Table 6.2, the actual calculation of the Cumulative Probability density function (CPDF) was shown for a selected scenario.

Figure 6.4 presents the result of the mutation analysis used to verify that the predictions, based on network analysis score, truly indicate functionally important methods. In Figure 6.4, the lines present the results for the four ranking methods (based on network measures HCS, WCS, BWS and NetBeans call frequency – CFS(p)) by showing the percentage of the ranked methods up to a given rank, which are experimentally checked to be functionally important methods. For example, 100% for a rank r, means that all methods up to rank r are functionally important, whilst 50% means that only half of them up to this rank are functionally important. The horizontal dashed line shows the percentage of randomly chosen methods that are functionally important. The vertical axis shows the probability of methods that are checked to be functionally important out of the top r ranked methods. The value of r is given by the horizontal axis.

As discussed in chapters 4 and 5, the mutations to the software were carried out by introducing syntactically and semantically correct stubs that provided shortcuts to the method's functionality. This is to simulate the condition of network edge removal. An error free compilation attempt is made to re-execute the scenarios with the new mutated software. If no effect of the change on normal operation of the software is experienced, the result is marked as "0". In the case that normal operation is partially or wholly affected the result is marked as "1".

In addition to the mutation tests on the lists of scores, another list is generated containing 101 methods chosen at random. These methods are also subjected to mutation testing. The scenarios used to test the mutant software can be chosen at random from one of the existing scenarios. It was found that 95% confidence interval, of the cumulative distribution function of the random tests, lie at 0.60 and 0.49 respectively for the upper and lower limits. The average was found to be approximately 0.55.

In Figure 6.4, the results of random analysis appear as black dashed lines denoting the upper and lower limits of the 95% confidence interval. The average of the distribution appears as a solid black line. The results for the analysis of random selection of methods provide a base line against which the results of mutation testing of network analysis ranked methods can be compared. This comparison is to show whether the performance of network analysis methods is better than by chance, or otherwise.

The results of validation of functional importance predictions shown in Figure 6.4 corresponds to the Top 20 methods predicted by the network analysis measures considered, namely: HCS,WCS, BWS and CFS(p). There is no particular reason to choose the first 20 highest ranked methods from the list, except the convenience to present the findings and also the thought that if the top 20 predictions are given to a developer it would represent a manageable number of items to verify, rather than a larger number. In this case, the logic of the 80-20 split, based on Pareto analysis rule of thumb, has not been used as that number would be 33 for 165 classes. The analysis has been done for the first 50 methods in all cases. The results indicate that HCS, WCS and

NetBeans call frequency rankings are more likely to be functionally important than randomly picked methods for the first 14-15 positions of the ranked lists. However, the functional importance prediction performance for these rankings is comparable or below the random chance for the ranks above 14/15 in the case of call frequency and HCS ranking, and above 20 in the case of WCS ranking. The ranking based on BWS metric does not predict important methods better than chance.

It can be seen that scores do not consider the method invocation frequency information CFS (m) or CFS (p) similar to the Hub connection score (HCS), and the Betweenness score (BWS) generally performs poorly. In contrast, scores that use method invocation frequency CFS (m), such as the WCS, demonstrates performance comparable to the profiler generated score CFS(p). However, results indicate that the top five methods indicated by CFS (p) are not found to be functionally important in the analysis, and this indicates that the top profiler invocation frequency ranks may not be a real indicator of the functional importance of methods.

6.3.2 Combined analysis of network measures

In Section 5.2.2, the scheme used to generate combined rankings based on the sum and product of network analysis scores for individual methods in separate ranked lists was discussed (see equations 5.11 and 5.12).

$$s(M1, M2) = r_1 + r_2 \quad (5.11)$$

$$s(M1, M2) = r_1 \times r_2 \quad (5.12)$$

In cases where the scores are comparable and normalisable, non parametric ranking approaches exist (e.g. Wilcoxon rank-sum/Mann-Whitney U test). However, in the case of network analysis, use of these established statistical techniques may not be appropriate, because the ranked lists based on network analysis scores are not comparable. However, linear transformation [234] can be applied as a standard technique to shift the search plane.This particular interpretation is in the context of vector space transformation, where some dimensions are nullified due to transformation. In Table 6.4, the top 2 vanilla ranked methods and rankings achieved by sum and product combination of network analysis scores are presented.The mutation analysis was performed to verify that combination rankings predict functionally important method calls. The mutation analysis technique followed in the case of combination ranking analysis is identical to the technique used for verification of vanilla rankings discussed in Chapter 4, 5 and in the previous section.

Figures 6.5-6.7 provide a graphical representation of the performance of the ranking methods considered. These combinations are: a) HCS-BWS, b) HCS-CFS(m), c) HCS-BWS-CFS(m), and d) BWS-CFS(m). In Figure 6.5, the results of considered rankings are derived by summation of the network analysis scores. In Figure 6.7, results of the considered rankings are derived by calculating the product of the network analysis scores. In Figure 6.6, the results presented exhibit the BWS-CFS(p) metrics for both sum and product operation derived rankings. In all, the figures present the result of mutation analysis of the 101 randomly selected methods of JHotdraw software. These appear as a continuous black line for the average value of randomly selected methods that appear to be functionally important. The dashed black lines correspond to the upper and lower bounds of the 95% confidence interval for the random selection test.

	HCS metric		**BWS metric**
Rank	**Method**	**Rank**	**Method**
1	standard.StandardDrawingView.tool	1	standard.StandardDrawingView.tool
2	standard.StandardDrawingView.paintComponent	2	framework.FigureAttributeConstant.getName
	CFS metric		**HCS + BWS combined metric**
Rank	**Method**	**Rank**	**Method**
1	framework.FigureAttributeConstant.hashCode	1	contrib.AutoscrollHelper.Constructor
2	figures.FigureAttributes.get	2	standard.StandardDrawingView.paintComponent
	HCS + CFS combined metric		**BWS + CFS combined metric**
Rank	**Method**	**Rank**	**Method**
1	framework.FigureAttributeConstant.hashCode	1	standard.AbstractCommand$1.viewSelectionChanged
2	standard.StandardDrawingView.tool	2	standard.AbstractFigure.invalidate
	HCS + BWS + CFS combined metric		**HCS * BWS combined metric**
Rank	**Method**	**Rank**	**Method**
1	standard.StandardDrawingView.tool	1	standard.StandardDrawingView.paintComponent
2	standard.AbstractCommand$1.viewSelectionChanged	2	contrib.AutoscrollHelper.Constructor
	HCS * CFS combined metric		**BWS * CFS combined metric**
Rank	**Method**	**Rank**	**Method**
1	framework.FigureAttributeConstant.hashCode	1	contrib.zoom.ZoomDrawingView$2.mouseMoved
2	standard.StandardDrawingView.tool	2	figures.AttributeFigure.getAttribute
	HCS * BWS * CFS combined metric		
Rank	**Method**		
1	standard.StandardDrawingView.tool		
2	standard.AbstractCommand$1.viewSelectionChanged		

Table 6.4 The top-2 ranked methods according to each individual and combined metric.

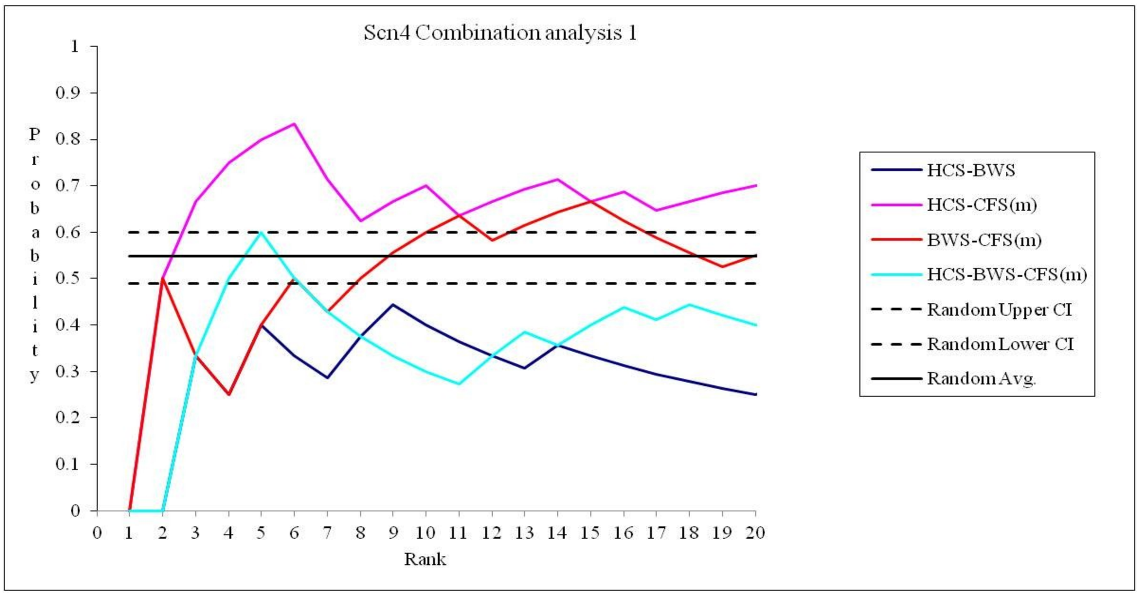


Figure 6.5 The performance of combination ranking from the summation of network analysis scores.

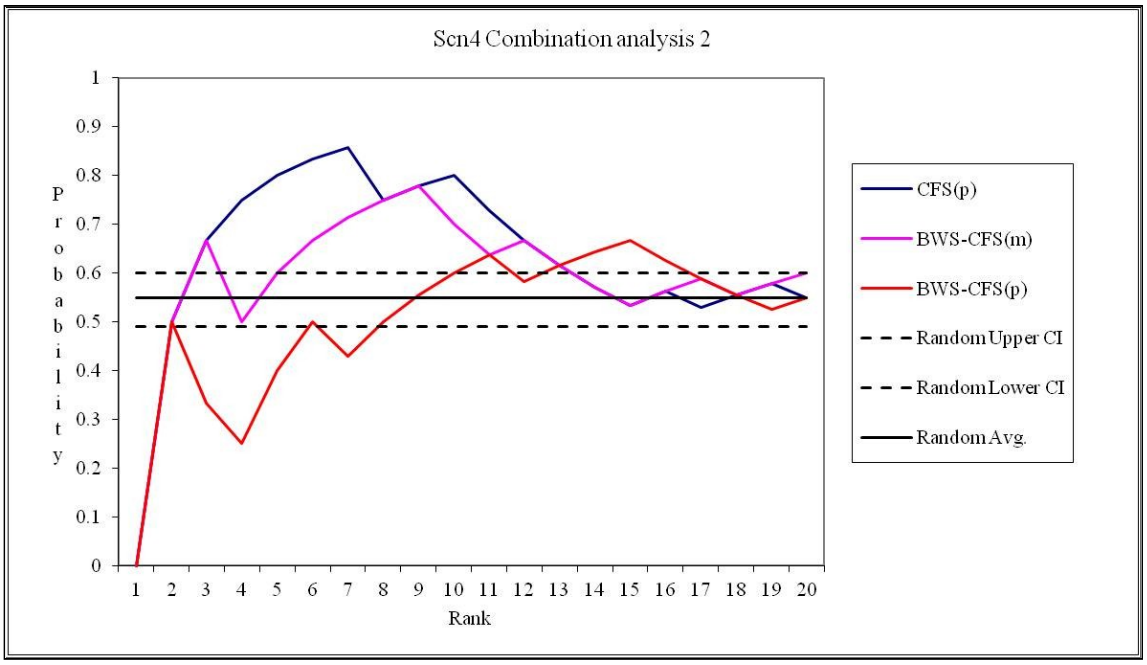


Figure 6.6 The performance of combination ranking from the summation and product of network analysis scores.

Figures 6.5-6.7 all present the performance of network analysis derived rank combination analysis. The solid back line in these figures represents the result of mutation analysis on a random selection of 100 methods, and the dashed black lines indicate the upper and lower bounds of the confidence interval for the mutation analysis on the randomly selected methods.

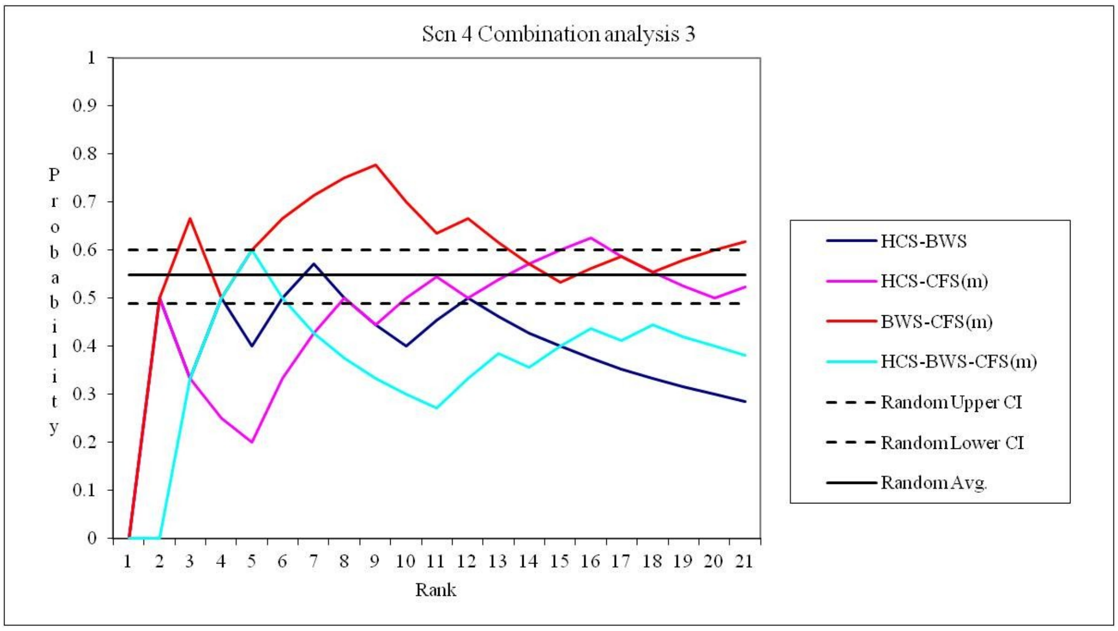


Figure 6.7 The performance of combination ranking from the product of network analysis scores.

6.4 Discussion

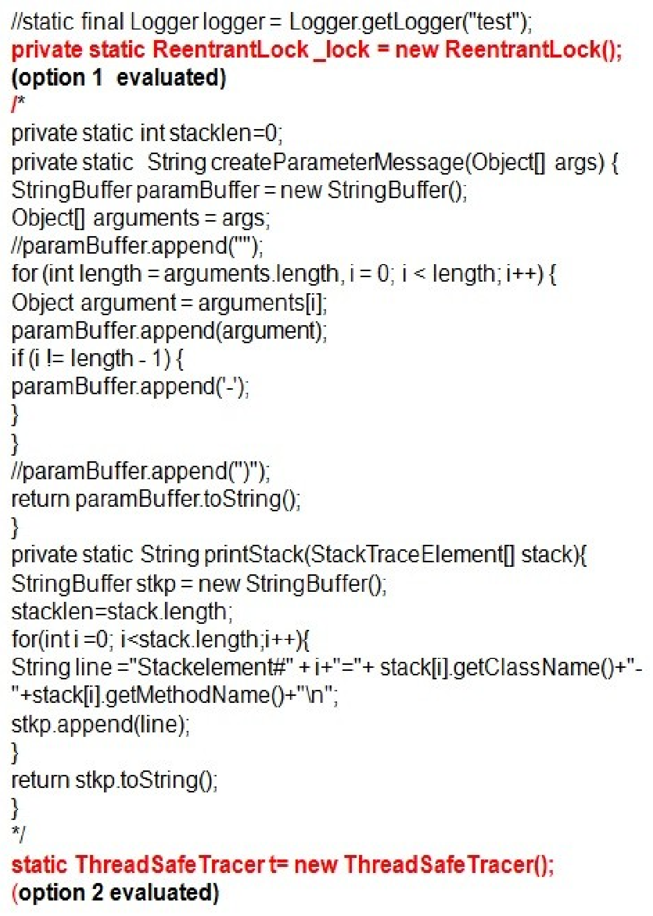
In Table 6.4, for the sake of brevity and clarity the top two methods appearing in each of the 11 ranked lists have been presented. This table presents rankings obtained by combining the results of rankings obtained by vanilla analysis. The scheme used to generate the combined ranking has been discussed in Chapter 5 and in the preceding section.

The performance of a network analysis metric, or of a metric combination, is calculated as the percentage of methods that are identified as important on the basis of the metric(s), and which turn out to be important according to the functional evaluation of methods. If all of the top-n ranked methods turn out to be functionally important then we have 100% performance for the ranking up to the n, and if only half of the top-n methods turn out to be functionally important then the performance of the ranking for n is 50%. The performance results are presented in Figures 6.5-6.7.

The results (see Figure 6.4) show that most of the top-15 ranked methods according to the HCS and CFS analysis metrics are highly functionally important, while this is not the case to the same extent for ranking of methods based on the BWS metric. In the case of combined metrics, the results show that the additive combination or rank orders did not produce better combined metrics than the individual metrics. However, in the case of rank product based combination of metrics, the performance of combined metrics improved for the top-50 methods, except for the metric combination HCS – BWS.

The results show that the HCS – BWS combination in both cases of rank combinations (sum and product) led to a combined ranking of methods that identified a smaller percentage of functionally important methods than the rankings based on the individual metrics. This indicates that these two metrics are likely to identify different kinds of methods as important, and combining the separate rankings in a sense cancels the correct identification of functionally important methods. Notably, the triple combination of rankings does not perform better than the paired combinations of rankings, which again is likely to be due to the cancellation effect of the combination of HCS and BWS rankings. It is also noticed that combinations which include CFS score tend to produce improved detection performance. It can also be seen in Table 6.4 that in some cases the result of combination rankings leads to utility methods rising to the top of the rankings, particularly when combinations include BWS score. The reciprocal element of invocation frequency, used to weigh edges on the shortest paths, seem to be leading to this phenomenon. These utility methods have lower invocation frequency, in comparison to utilities that belong to the tool bar classes. For instance the auto scroll helper method is only likely to be invoked in the event of zoom tool operation, to enable scrolling in the drawing pane. Its functional significance in the context of overall functional delivery is negligible in the context of the definition of functional importance used in these experiments.

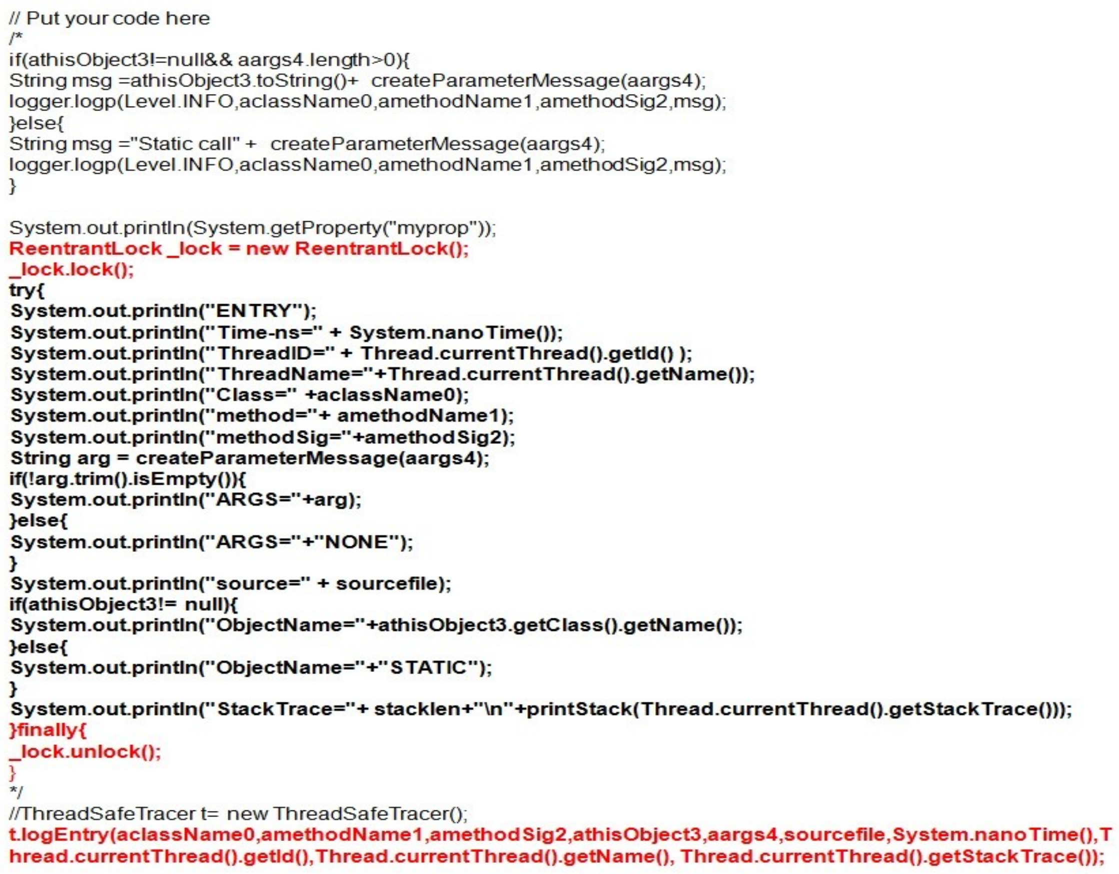
In experiments of this nature, many scenario runs are typically required to quantify the statistical significance of results with a reasonable degree of confidence. Random tests alone provide some confidence, but are generally considered weak. A lesson drawn from these experiments concerns the fragility of Eclipse TPTP tools, and tools generally built with Java BECL, that underpin both static and dynamic instrumentation frameworks. The problems encountered are two-fold: a) the upgrades of JVM classes related to concurrency, and b) TPTP project's inability to maintain reliability. Through most of this research, the instrumentation technique of choice has been TPTP Probekit. This tool (discussed in Chapter 5) has static binary instrumentation capability and provides a user friendly interface that facilitates definition of custom probes. The versions of TPTP and JVM used during the initial proof of concept phase of these experiments were 4.5.5 and 1.6.0. At that point, all traces generated were verified for structural sanity, but the TPTP tool had stability problems. The later version (4.6.2) was relatively more stable, but it had introduced a subtle bug related to thread safety due to changes in JVM. This bug is particularly severely manifested in cases of large applications that make intensive use of threading. In this experiment, 60 long scenarios were generated using this bug hampered version. The bug introduced interleaving and overlapping of sections of the execution trace. This problem was identified and mitigation efforts did not lead to desirable results. Figures 6.8 and 6.9 provide illustrations of mitigation efforts which attempted to resolve thread safety issues noticed in Probekit. Figure 6.8 is an illustration of the use of ThreadSafeTracer library and Figure 6.9 illustrates the inline use of the "ReentrantLock()" function in the body of the TPTP probe.



```
//static final Logger logger = Logger.getLogger("test");
private static ReentrantLock _lock = new ReentrantLock();
(option 1  evaluated)
/*
private static int stacklen=0;
private static  String createParameterMessage(Object[] args) {
StringBuffer paramBuffer = new StringBuffer();
Object[] arguments = args;
//paramBuffer.append("");
for (int length = arguments.length, i = 0; i < length; i++) {
Object argument = arguments[i];
paramBuffer.append(argument);
if (i != length - 1) {
paramBuffer.append('-');
}
}
//paramBuffer.append(")");
return paramBuffer.toString();
}
private static String printStack(StackTraceElement[] stack){
StringBuffer stkp = new StringBuffer();
stacklen=stack.length;
for(int i =0; i<stack.length;i++){
String line ="Stackelement#" + i+"="+ stack[i].getClassName()+"-
"+stack[i].getMethodName()+"\n";
stkp.append(line);
}
return stkp.toString();
}
*/
static ThreadSafeTracer t= new ThreadSafeTracer();
(option 2 evaluated)
```

Figure 6.8 TPTP Probes using ThreadSafeTracer library.

The advantage of Probekit, apart from custom probe insertion, is that it is able to capture every event of interest, which may not be the case with TPTP Agent profiler. Custom structured traces also produce relatively smaller trace files compared to Agent profiler generated traces in the XML4Profiling format. The sanity checks found that later files had between 3-8% interleaved event writing, with errors increasing as the number of active threads increased in the application. In the case of Probekit generated traces, due to the custom structure and lack of redundant information or tags, a trace sanity checker was able to write to verify trace quality. However, with TPTP traces that can often be large (e.g. multi-gigabyte files) writing a sanity checker may not be easy. Both Agent profiler and Probekit use the same underlying framework, which raises concerns about the quality of TPTP profiler generated traces too.



```
// Put your code here
/*
if(athisObject3!=null&& aargs4.length>0){
String msg =athisObject3.toString()+  createParameterMessage(aargs4);
logger.logp(Level.INFO,aclassName0,amethodName1,amethodSig2,msg);
}else{
String msg ="Static call" +  createParameterMessage(aargs4);
logger.logp(Level.INFO,aclassName0,amethodName1,amethodSig2,msg);
}

System.out.println(System.getProperty("myprop"));
ReentrantLock _lock = new ReentrantLock();
_lock.lock();
try{
System.out.println("ENTRY");
System.out.println("Time-ns=" + System.nanoTime());
System.out.println("ThreadID=" + Thread.currentThread().getId() );
System.out.println("ThreadName="+Thread.currentThread().getName());
System.out.println("Class=" +aclassName0);
System.out.println("method="+ amethodName1);
System.out.println("methodSig="+amethodSig2);
String arg = createParameterMessage(aargs4);
if(!arg.trim().isEmpty()){
System.out.println("ARGS="+arg);
}else{
System.out.println("ARGS="+"NONE");
}
System.out.println("source=" + sourcefile);
if(athisObject3!= null){
System.out.println("ObjectName="+athisObject3.getClass().getName());
}else{
System.out.println("ObjectName="+"STATIC");
}
System.out.println("StackTrace="+ stacklen+"\n"+printStack(Thread.currentThread().getStackTrace()));
}finally{
_lock.unlock();
}
*/
//ThreadSafeTracer t= new ThreadSafeTracer();
t.logEntry(aclassName0,amethodName1,amethodSig2,athisObject3,aargs4,sourcefile,System.nanoTime(),T
hread.currentThread().getId(),Thread.currentThread().getName(), Thread.currentThread().getStackTrace());
```

Figure 6.9 TPTP probe with inline use of ReentrantLock().

In experiments of this nature, it is important that not only the results of the experiments are statistically significant, but also that the trace data itself is largely error free.

Effort has been made to improve the quality of statistical treatment in the experimental results in Chapter 7, which builds on the early work presented here, albeit with a very large C++ software system.

Chapter 7. Experiments on Google Chrome

7.1 Introduction

Google Chrome [210] is a popular open source web browser, managed by Google and supported by an active development community. The Google Chrome project provides open and free access to development documentation and source code, making it an attractive choice for empirical software engineering experiments. In Chapter 4, the concept of functional importance was proposed and defined, and a technique to verify the functional importance of methods was discussed. In Chapter 5, the network analysis measures used to predict functionally important methods were discussed as part of the experiment design, and the verification technique was discussed. A key assumption behind the network based analysis used in this thesis is that object oriented software networks follow the power law connectivity distribution. In this context, it was also noted in Chapter 5 that power law distribution is difficult to identify, and often distributions fall in the border-line cases due to fat tail distribution characteristics. In Chapter 6, we found that the network representation of JHotdraw (version 6.0b1) displayed the log-normal distribution instead of the pure power law distribution. This chapter is focused on presenting the results from the experiments on Google Chrome. These experiments used custom profiler generated dynamic analysis data and the experiment techniques discussed in Chapter 5. The only key variation in the workflow compared with the experiments on JHotdraw is the use of Cloud based mutation testing for verification [14].

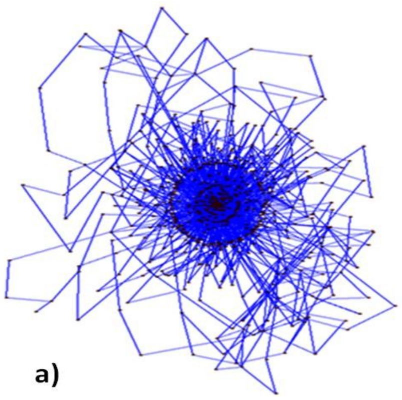
7.1.1 Google Chrome

The version of Google Chrome used for this experiment was 72930. The total source lines of code (SLOC) was found to be 8,942,123 – spread over 480 Microsoft Visual studio projects and 9 source code packages: base, chrome, content, media, native_client, net, printing, remoting and webkit. The lines of code (LOC) was calculated using UNIX tool SLOC-Count [235]. Google Chrome incorporates many advanced and novel features, such as support for HTML5, native client execution, and a highly optimized java script engine and compiler called V8. The design objectives behind this browser were motivated by the need to implement better program abstraction within web browser application for improved performance [236]. Notice that the choice of Google Chrome for this experiment is not based on comparative evaluation of comparable browser applications, since we believe that every application is likely to display a unique signature from complex network analysis. Google Chrome was chosen because it is professionally designed object oriented software, which is open source and available on a wide range of OS platforms. The availability on a wide range of platforms makes it attractive for use in software engineering experiments.

The use of Google Chrome was found to be challenging, because of the custom built system it uses. The 'Generate your projects' (GYP) scripts that are used to build this complex application are not easy to modify for experimental purposes, such as static instrumentation based on compiler features. Execution trace profile data was collected by using a modified version of the Slimtune [225] tool. This tool is based on Microsoft Visual Studio profiler and dynamically instruments Google Chrome to collect program execution trace data.

7.2 Data from execution profile trace

In Chapter 5, the technique used to convert execution profile trace information into the graph representation containing objects as nodes and method calls as edges was discussed. The same technique was employed in the experiments on Google Chrome, the only difference being the profile trace data was stored in a SQLlite database file by Slimtune profiler. This network representation has been visualized (see Figure 7.1 a) using the Pajek tool [229]. Figure 7.1b presents the cumulative node connectedness distribution of the network representation of dynamic analysis data, as generated by the execution of the Google Chrome. The linear relationship between the log(Connectedness) and log(CumFreq) indicates that the distribution follows a power law.



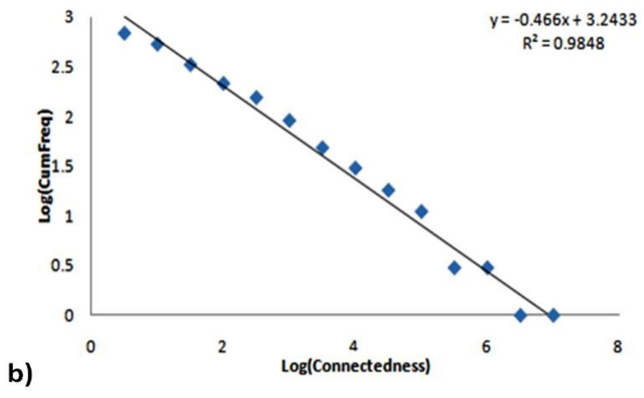


Figure 7.1 The network representation of Google Chrome.

The power law distribution was verified by analysing the cumulative connectedness distribution of the network nodes (the connectedness of a node is the number of edges connecting a node to other nodes). Considering all edges for all nodes representing classes, the best fitting distribution is an integrated power law distribution (see Figure 7.1 b) with the cumulative probability density function of the connectedness values given as

$$p(x \geq a) = e^{3.2433} \cdot a^{1-1.466} \quad (7.1)$$

This distribution (Eqn. 7.1) satisfies the general requirement for the application of network analysis methods. That is, the network should have a power law node connectedness distribution. In Chapter 5, the discussion of power law verification mentioned various research publications, proposing alternate methods to verify power law distribution in network graph data. The technique used in this thesis is appropriate for our purposes, and application of network analysis can be used to predict the functionally important methods in Google Chrome.

7.3 Network Analysis of data

The network analysis measures (from Chapter 5) used to analyse Google chrome network are:

- Weighted connection score (WCS)

$$WCS\ (m) = f(e).v(n).v(m) = HCS(e).f(e) = \sum_{n} f(e|n).v(n) \quad (5.7)$$

- The total call frequency score (CFS(m))

$$CFS(m) = \sum_{n} f(e|n) \quad (5.8)$$

The method's invocation frequency score was captured by the Slimtune profiler tool in the traces.

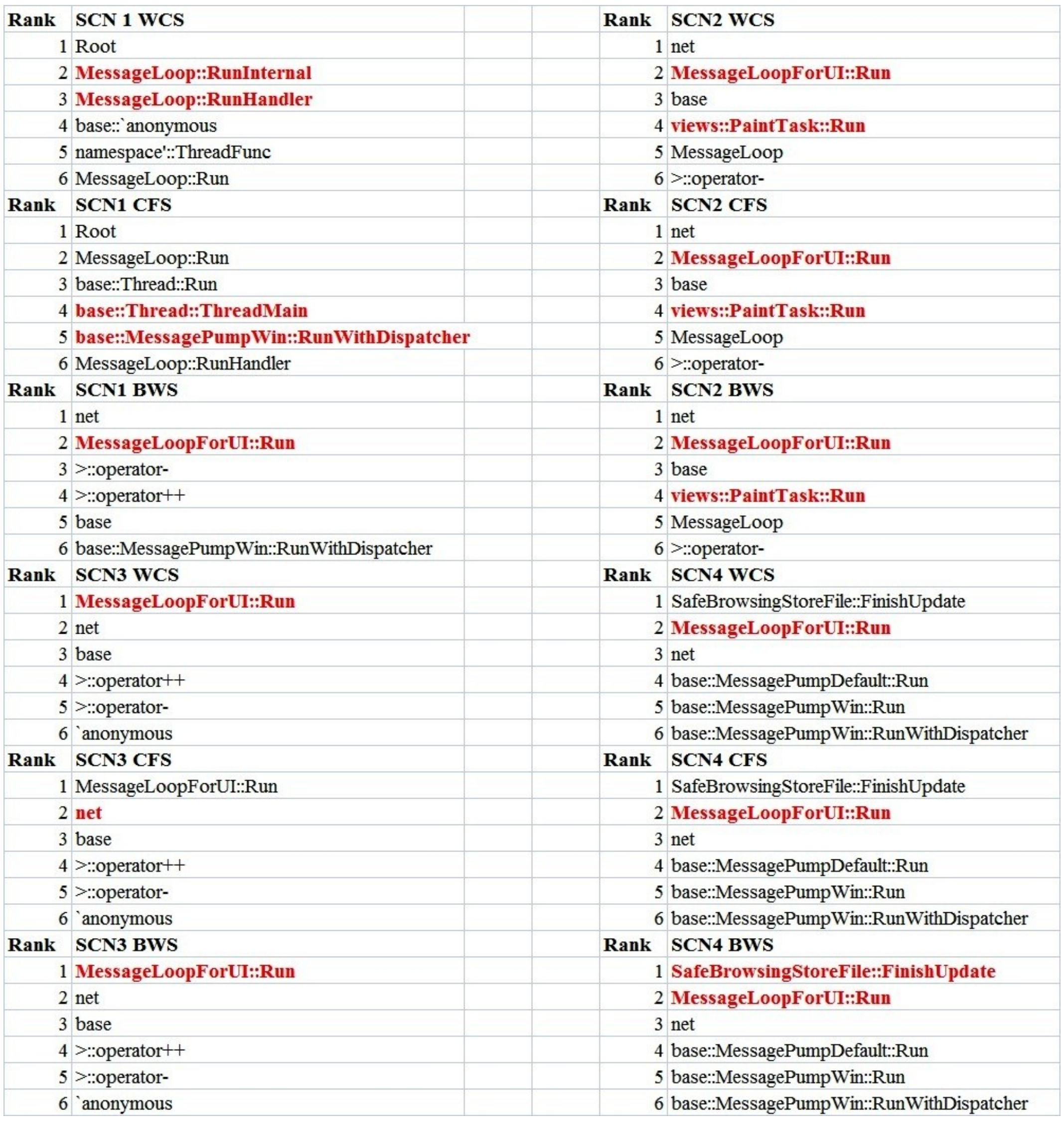
- Betweenness score (BWS)

The Betweenness score is the only measure that requires the computation of shortest paths, and the technique used to calculate the shortest paths, as discussed in Chapter 5 and Chapter 6, is based on use of Dijkstra's algorithm [78].

$$BWS(m) = \max_e \left| \begin{array}{c} \{e_1, \dots, e_k\} | e \in \{e_1, \dots, e_k\}, \\ \sum_{i=1}^{k} \frac{1}{f(e_i)} \leq \sum_{j=1}^{k'} \frac{1}{f(e'_j)}, \\ \forall \{e'_1, \dots, e'_{k'}\}: \\ |nodes(e_1) \cap nodes(e'_1)| \geq 1, \\ |nodes(e_k) \cap nodes(e'_{k'})| \geq 1, \\ |nodes(e_i) \cap nodes(e_{i+1})| = 1, \\ |nodes(e'_j) \cap nodes(e'_{j+1})| = 1, \\ i = 1, \dots k;\ j = 1, \dots, k' \end{array} \right| \quad (5.10)$$

$$where\ nodes(e)\ determines\ the\ two\ nodes\ connected\ by\ edge\ e$$

For brevity, Table 7.1 presents the top 6 predictions for all the scores for 4 scenarios.The predictions generated by all the network measures have been presented in this table. The validation included the top 50 predictions.



Rank	SCN 1 WCS	Rank	SCN2 WCS
1	Root	1	net
2	**MessageLoop::RunInternal**	2	**MessageLoopForUI::Run**
3	**MessageLoop::RunHandler**	3	base
4	base::`anonymous	4	**views::PaintTask::Run**
5	namespace'::ThreadFunc	5	MessageLoop
6	MessageLoop::Run	6	>::operator-
Rank	**SCN1 CFS**	**Rank**	**SCN2 CFS**
1	Root	1	net
2	MessageLoop::Run	2	**MessageLoopForUI::Run**
3	base::Thread::Run	3	base
4	**base::Thread::ThreadMain**	4	**views::PaintTask::Run**
5	**base::MessagePumpWin::RunWithDispatcher**	5	MessageLoop
6	MessageLoop::RunHandler	6	>::operator-
Rank	**SCN1 BWS**	**Rank**	**SCN2 BWS**
1	net	1	net
2	**MessageLoopForUI::Run**	2	**MessageLoopForUI::Run**
3	>::operator-	3	base
4	>::operator++	4	**views::PaintTask::Run**
5	base	5	MessageLoop
6	base::MessagePumpWin::RunWithDispatcher	6	>::operator-
Rank	**SCN3 WCS**	**Rank**	**SCN4 WCS**
1	**MessageLoopForUI::Run**	1	SafeBrowsingStoreFile::FinishUpdate
2	net	2	**MessageLoopForUI::Run**
3	base	3	net
4	>::operator++	4	base::MessagePumpDefault::Run
5	>::operator-	5	base::MessagePumpWin::Run
6	`anonymous	6	base::MessagePumpWin::RunWithDispatcher
Rank	**SCN3 CFS**	**Rank**	**SCN4 CFS**
1	MessageLoopForUI::Run	1	SafeBrowsingStoreFile::FinishUpdate
2	**net**	2	**MessageLoopForUI::Run**
3	base	3	net
4	>::operator++	4	base::MessagePumpDefault::Run
5	>::operator-	5	base::MessagePumpWin::Run
6	`anonymous	6	base::MessagePumpWin::RunWithDispatcher
Rank	**SCN3 BWS**	**Rank**	**SCN4 BWS**
1	**MessageLoopForUI::Run**	1	**SafeBrowsingStoreFile::FinishUpdate**
2	net	2	**MessageLoopForUI::Run**
3	base	3	net
4	>::operator++	4	base::MessagePumpDefault::Run
5	>::operator-	5	base::MessagePumpWin::Run
6	`anonymous	6	base::MessagePumpWin::RunWithDispatcher

Table 7.1. Top-6 predictions from network analysis of Google Chrome.

7.3.1 Use of the cloud in mutation testing work flow

In the context of this experiment, a key challenge in performing mutation testing to verify the predictions of functional importance presented in Table 7.1 was the size and build complexity of Google Chrome. This led to the requirement to build multiple manually prepared mutants of Google Chrome based on our prediction in the Cloud. As part of the experiments, the mutation build process was automated to enable the mutation testing process to be performed efficiently.

The data collection platform in our experiment (which uses the technique and workflow discussed in Chapter 5, Section 5.2.5 and Section 5.2.7) was Microsoft Windows 7 Professional 64 bit. The user driven test scenarios were generated by using data from Google Chrome Windows browser application. The scenarios used were similar to the user acceptance testing scenarios used for web browser applications that simulate the typical use of web browser applications.

The natural choice of platform for building mutants would have been Microsoft Windows 7 Professional 64 bit. However, we found that neither Amazon nor Microsoft Azure services offer Microsoft Windows 7 Professional 64 bit as a choice. In this experiment, only function calls within Google Chrome are considered, which in theory means the application can be built on another platform, based on the assumption that non-system higher level method calls are common for all platforms. From the analysis point of view, this introduces a threat to validity. However, as far as the use of the Cloud is concerned, the process is valid. In this experiment, the Cloud instances used to build Google Chrome mutants were based on the Ubuntu 10.04.1 64 bit platform.

The experiments to verify predictions of functional importance for methods in Google Chrome required the building of 820 mutants. A scheme was implemented to manage all the data, so that for each mutant the input and output data could be isolated. This was done in the first instance by using an unambiguous naming scheme for the mutants. This mutant test naming scheme was automated in the workflow script. A careful scan of the data was carried out for all the scenarios to ensure that no duplicate mutant was prepared. A large number of common elements occurred across predictions, and in all these cases only one mutant was prepared, and the analysis log was used in the instance for which the mutant was previously generated. In this way, 820 builds were reduced to 186. This is approximately a 75% reduction in initial estimation of costs. The total input data across all builds was approximately 100 Gigabytes (GB), and the output data, consisting of only the built executable and supporting libraries, was 52 Gigabytes (GB).

The total computation time, as per Amazon, was approximately 561 hours, which represents a 25% reduction over the 744 hours it would have taken to run the same test using the local workstation. These figures fail to point out that 561 hours is actually clocked in 7 days of test runs and includes some repeats. 744 hours on a local workstation would have taken 31 days with the building process running 24 hours a day. The compressed input data of 0.5 GB expands during build time to around 15GB each, which would require about 2.7 Terabytes (TB) of storage. The total charge for use of the Amazon Cloud for this experiment is approximately $1000.00. This includes the use of data transfer, use of S3 storage, and m2.xlarge EC2 instances. The cost advantage of using the Cloud is amply clear, in addition to the time efficiency related saving. If cost and time saving are the main factors that influence the use of Cloud based testing, this presents a clear advantage over captive storage and computational infrastructure. However, these may not be the main influencing factors. In our view, the main factors are flexibility of configuration and the use of a test platform. The workflow adopted to implement the Cloud based mutation testing workflow [14] is presented in Figure 7.2.

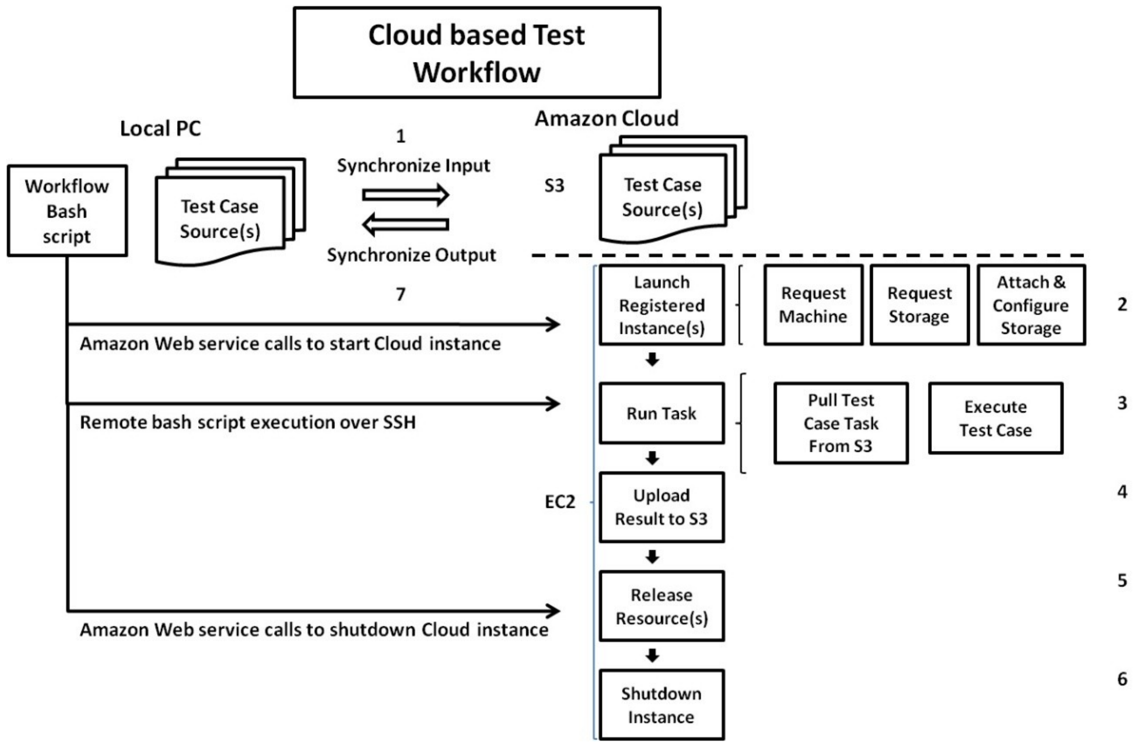
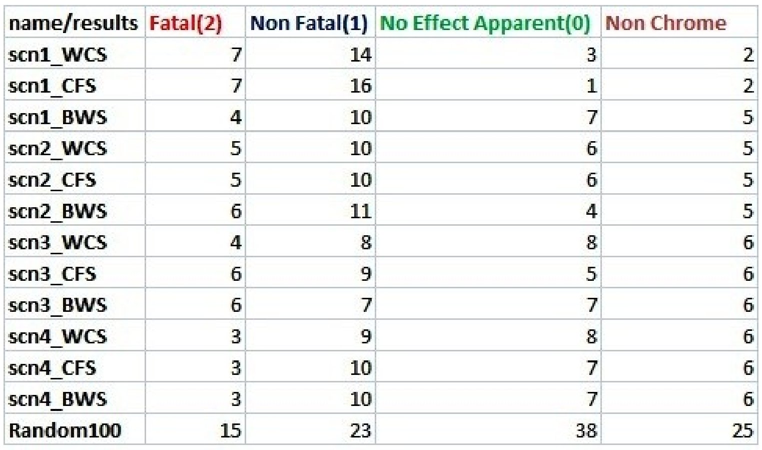

Cloud based Test Workflow
Local PC
Amazon Cloud
Workflow Bash script
Test Case Source(s)
1
Synchronize Input
Synchronize Output
7
S3
Test Case Source(s)
Launch Registered Instance(s)
Request Machine
Request Storage
Attach & Configure Storage
2
Amazon Web service calls to start Cloud instance
Remote bash script execution over SSH
Run Task
Pull Test Case Task From S3
Execute Test Case
3
EC2
Upload Result to S3
4
Release Resource(s)
5
Amazon Web service calls to shutdown Cloud instance
Shutdown Instance
6

Figure 7.2 Schematic diagram of the Cloud-based mutation testing.

A selection of results from the mutation testing spanning all the scenarios is presented in Table 7.2. The results are tabulated into 3 outcomes of the mutation: a) software crash (i.e. fatal), b) runtime state corruption (i.e. non-fatal), and c) the mutation has no noticeable effect on software operation. Table 7.3 is an illustration of the data analysis steps used to calculate the probability density function, and the 95% confidence interval from the result data set. The statistical analysis steps have been labelled at the bottom of the table. The equation used to calculate confidence intervals is

$$CI \;=\; \overline{x} \pm 1.96 \frac{\sigma}{\sqrt{n}} \quad (7.2)$$

In this case, $\overline{x}$ is the average of the cumulative probability density function for chosen rank. Number n is sample size, where in this case it is 4; corresponding to 4 scenario data sets.



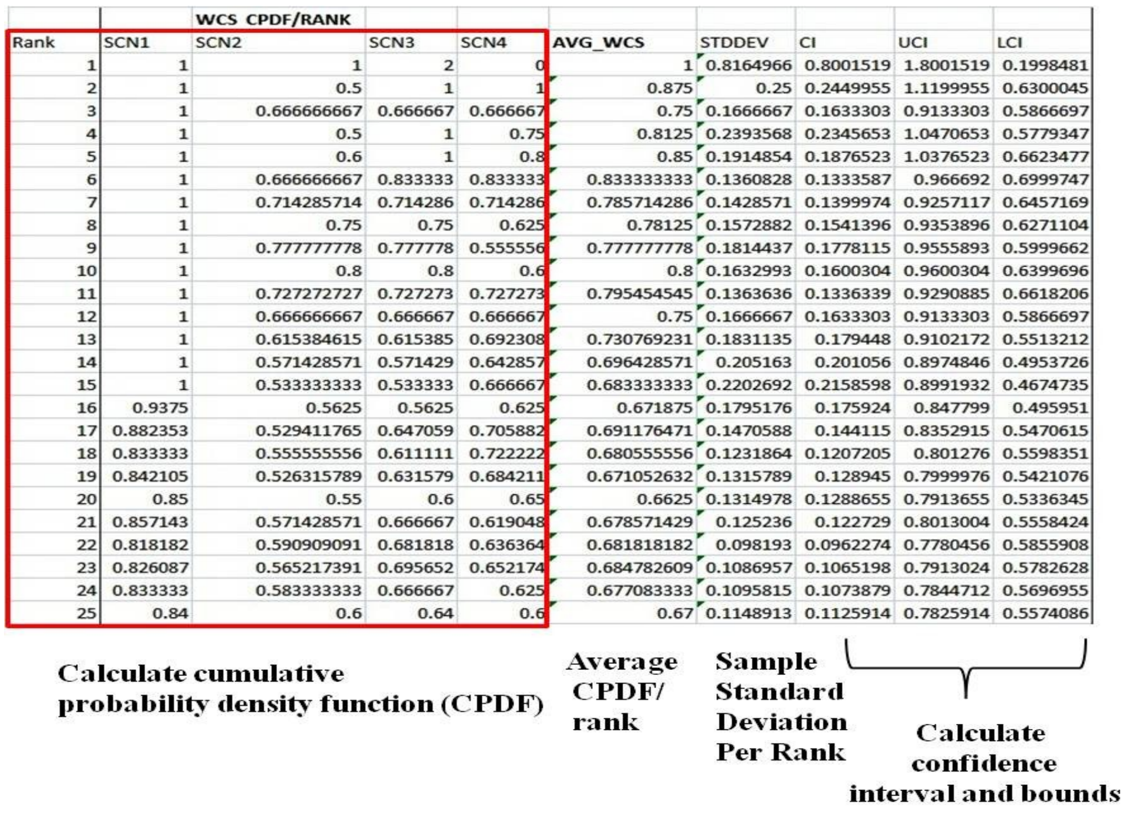
name/results	Fatal(2)	Non Fatal(1)	No Effect Apparent(0)	Non Chrome
scn1_WCS	7	14	3	2
scn1_CFS	7	16	1	2
scn1_BWS	4	10	7	5
scn2_WCS	5	10	6	5
scn2_CFS	5	10	6	5
scn2_BWS	6	11	4	5
scn3_WCS	4	8	8	6
scn3_CFS	6	9	5	6
scn3_BWS	6	7	7	6
scn4_WCS	3	9	8	6
scn4_CFS	3	10	7	6
scn4_BWS	3	10	7	6
Random100	15	23	38	25

Table 7.2 Tabulation of the results of verification of predictions of functional importance of Google Chrome.

In Table 7.3, computation of the probability density function is based on the verification of function importance using the mutation tests presented in Table 7.2. In Table 7.3, for each rank position all fatal and non-fatal results are marked "1" and results having no effect are marked "0". Non Chrome entries are rejected. This scheme is identical to the verification and statistical analysis of results presented in Chapter 6, which presents the results of experiments on JHotdraw 6.0b1. The only difference in presentation of the results arises because the results presented for JHotdraw were based on one scenario because of data collection errors, which compromised the quality of the scenarios collected using Eclipse TPTP Probekit (discussed in Chapter 6). Data from the JHotdraw scenario run Scn4 was found to have no errors and to be verifiable. The final verification graphs did not include calculation of standard deviation, except for the mutation testing of 10 randomly selected JHotdraw methods. In the case of Google

Chrome, the data collection tool was reliable and all four scenes have been included in the statistical analysis. This made it possible to calculate the standard deviation as illustrated in Table 7.3. The other difference in the presentation of results for Google Chrome is that only vanilla analysis is discussed, since no combination analysis was performed on the Google Chrome data. This is because (based on the results of experiments on JHotdraw) combination analysis did not contribute to the verification of functional importance in a substantial way and the techniques may require further refinement.

In Table 7.3, the statistical analysis performed for all 3 network measures is illustrated, and is considered to validate the predictions of functional importance. In the table, for the sake of conciseness, only the case of WCS measure is presented. The red box presents the cumulative probability density function (CPDF) per rank of the scores table. The data here has 25 rows for 4 scenes. The Average WCS CPDF per rank is calculated from this data. The standard deviation is also calculated. This is used to calculate the 95% confidence interval. These results follow from analysis of the data behind Table 7.2.


Calculate cumulative probability density function (CPDF)
Average CPDF/ rank
Sample Standard Deviation Per Rank
Calculate confidence interval and bounds

	WCS CPDF/RANK								
Rank	SCN1	SCN2	SCN3	SCN4	AVG_WCS	STDDEV	CI	UCI	LCI
1	1	1	2	0	1	0.8164966	0.8001519	1.8001519	0.1998481
2	1	0.5	1	1	0.875	0.25	0.2449955	1.1199955	0.6300045
3	1	0.666666667	0.666667	0.666667	0.75	0.1666667	0.1633303	0.9133303	0.5866697
4	1	0.5	1	0.75	0.8125	0.2393568	0.2345653	1.0470653	0.5779347
5	1	0.6	1	0.8	0.85	0.1914854	0.1876523	1.0376523	0.6623477
6	1	0.666666667	0.833333	0.833333	0.833333333	0.1360828	0.1333587	0.966692	0.6999747
7	1	0.714285714	0.714286	0.714286	0.785714286	0.1428571	0.1399974	0.9257117	0.6457169
8	1	0.75	0.75	0.625	0.78125	0.1572882	0.1541396	0.9353896	0.6271104
9	1	0.777777778	0.777778	0.555556	0.777777778	0.1814437	0.1778115	0.9555893	0.5999662
10	1	0.8	0.8	0.6	0.8	0.1632993	0.1600304	0.9600304	0.6399696
11	1	0.727272727	0.727273	0.727273	0.795454545	0.1363636	0.1336339	0.9290885	0.6618206
12	1	0.666666667	0.666667	0.666667	0.75	0.1666667	0.1633303	0.9133303	0.5866697
13	1	0.615384615	0.615385	0.692308	0.730769231	0.1831135	0.179448	0.9102172	0.5513212
14	1	0.571428571	0.571429	0.642857	0.696428571	0.205163	0.201056	0.8974846	0.4953726
15	1	0.533333333	0.533333	0.666667	0.683333333	0.2202692	0.2158598	0.8991932	0.4674735
16	0.9375	0.5625	0.5625	0.625	0.671875	0.1795176	0.175924	0.847799	0.495951
17	0.882353	0.529411765	0.647059	0.705882	0.691176471	0.1470588	0.144115	0.8352915	0.5470615
18	0.833333	0.555555556	0.611111	0.722222	0.680555556	0.1231864	0.1207205	0.801276	0.5598351
19	0.842105	0.526315789	0.631579	0.684211	0.671052632	0.1315789	0.128945	0.7999976	0.5421076
20	0.85	0.55	0.6	0.65	0.6625	0.1314978	0.1288655	0.7913655	0.5336345
21	0.857143	0.571428571	0.666667	0.619048	0.678571429	0.125236	0.122729	0.8013004	0.5558424
22	0.818182	0.590909091	0.681818	0.636364	0.681818182	0.098193	0.0962274	0.7780456	0.5855908
23	0.826087	0.565217391	0.695652	0.652174	0.684782609	0.1086957	0.1065198	0.7913024	0.5782628
24	0.833333	0.583333333	0.666667	0.625	0.677083333	0.1095815	0.1073879	0.7844712	0.5696955
25	0.84	0.6	0.64	0.6	0.67	0.1148913	0.1125914	0.7825914	0.5574086

Table 7.3 Probability density and confidence interval calculation of WCS score results.

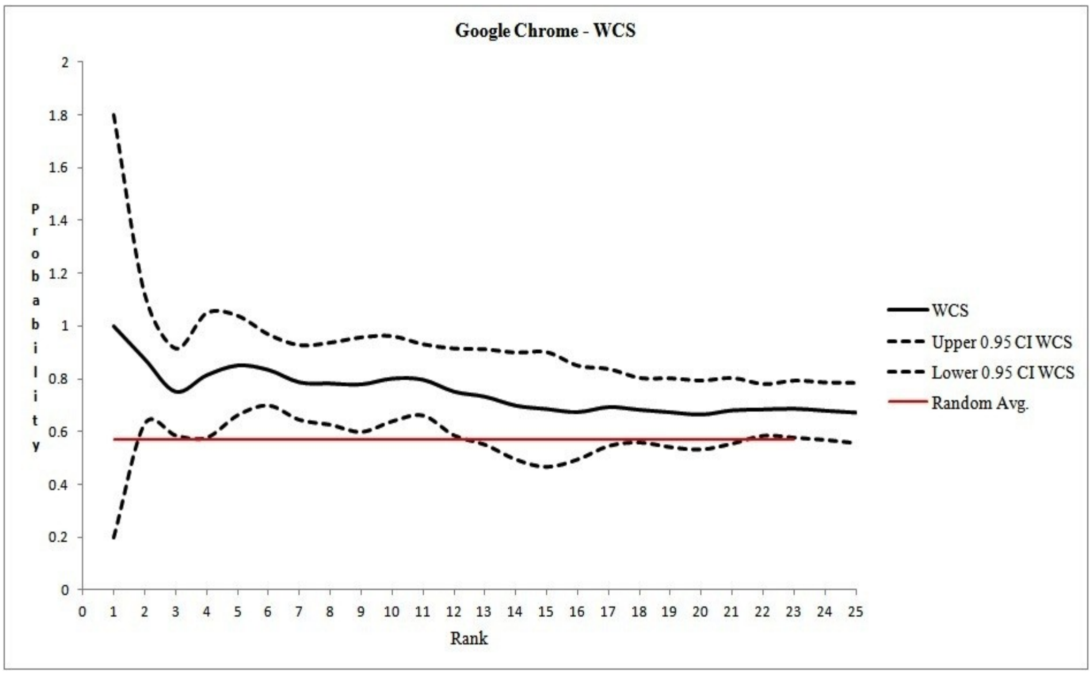


Figure 7.3 The performance of the vanilla ranking of predictions of functional importance of methods of objects, produced by WCS scoring.

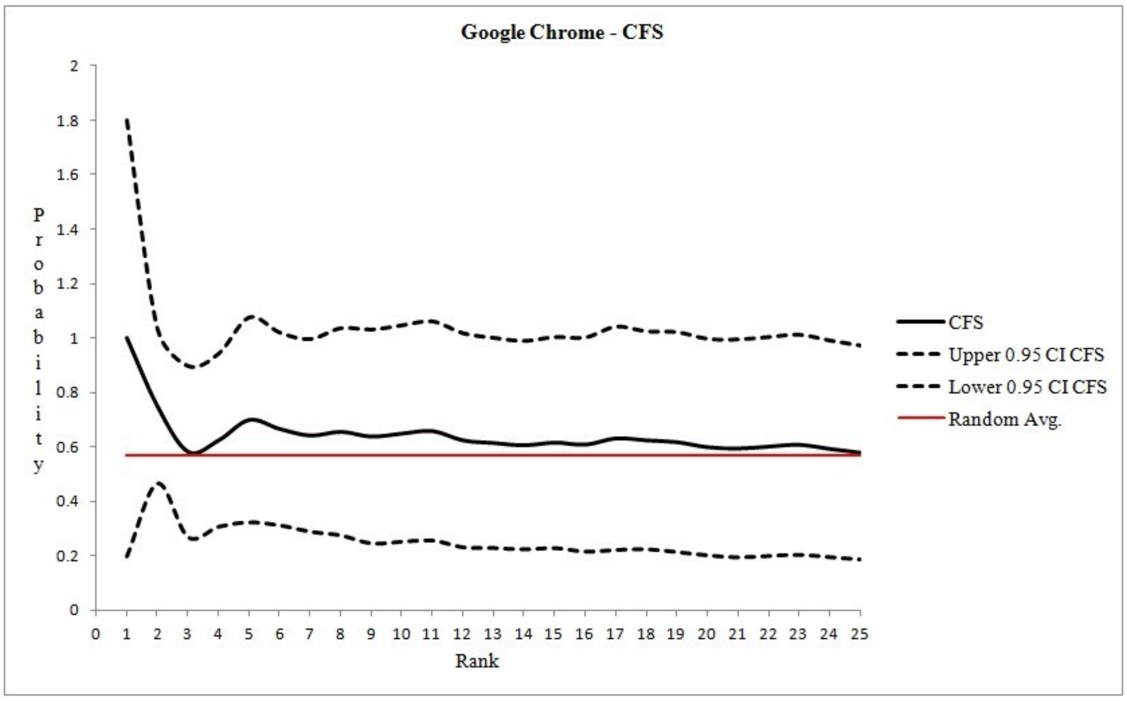


Figure 7.4 The performance of the vanilla ranking of predictions of functional importance of methods of objects, produced by CFS scoring.

The graphs for the results of verification of functional importance based on vanilla ranking produced by WCS, CFS(m) and BWS measures are presented in Figures 7.3-7.5. In these graphs, the top 25 methods from the ranked list are presented. The dashed lines show the 95% confidence bounds surrounding the average value and the red line shows the expected percentage of functionally important methods among a randomly chosen set of methods (50%).

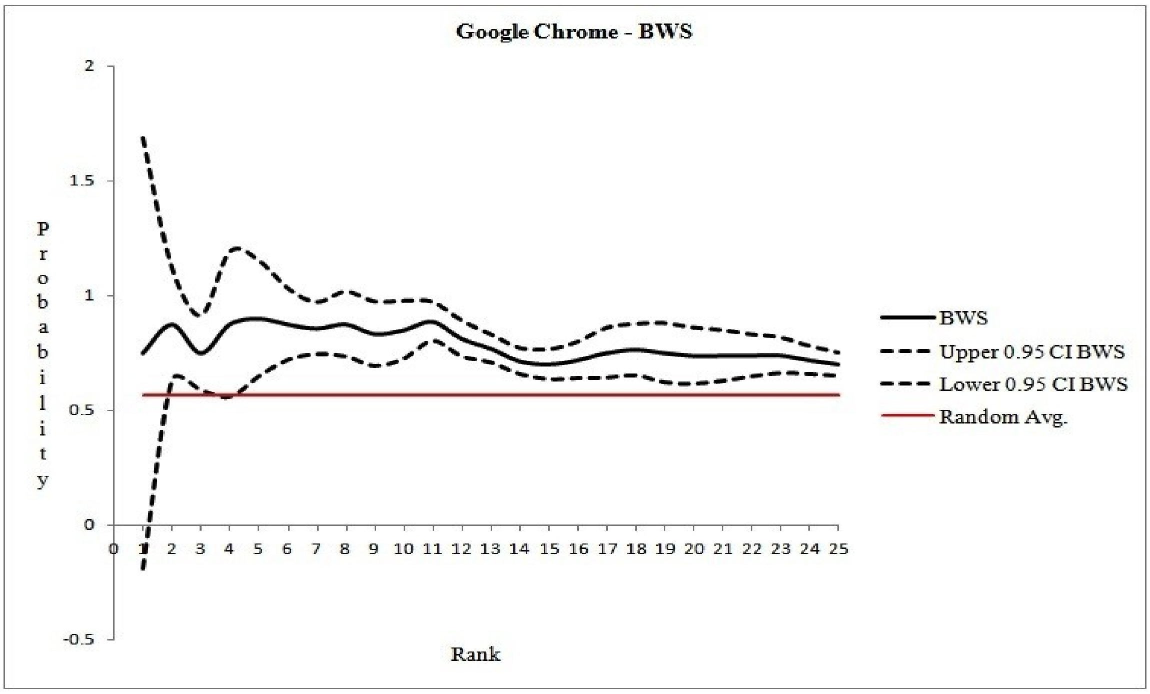


Figure 7.5 The performance of the vanilla ranking for predictions of functional importance in methods of objects produced by BWS scoring.

The results are presented in Figures 7.3-7.5. These show that all three network analysis based edge importance metrics that we considered are indeed better than chance in predicting methods of classes that are functionally important for the considered software functionality. In all cases, the 95% confidence band around the average prediction of accuracy level for all top-25 ranking levels is such that the 50% benchmark level is below the lower limit of the confidence band (note that the top ranked method is not always functionally important, causing the confidence bound for the top-1 method to drop below the 50% mark). In Figure 7.4, the line showing the random average is in the middle of the confidence bound, indicating that the method used for this figure does not produce results that are significantly better than the random choice of the methods. This means that the top-25 methods, selected according to the considered network structural importance metrics, in all cases are more likely to be functionally important than methods in a randomly picked set of methods, with the exception of CFS score as presented in Figure 7.4. Even in this case, for the top-25, the results show a marginally better performance than chance. This means that the network analysis of the network representation of the software system, based on dynamic analysis data, can be used to predict the functionally important methods and thus help the prioritisation of program comprehension, software testing activities, and the evaluation and potential improvement of its dependability.

7.4 Discussion

The results indicate, in the proof-of-concept sense, that network analysis measures of an object-method dynamic analysis network, when applied to large-scale software systems, can be used to predict successfully the functionally important methods of objects, and consequently classes, of an OO system, with respect to a given functionality of the software system.

Figure 7.6 is a diagram of the high level multi-process architecture of the Google Chrome browser. The diagram shows 3 blocks or high level components that, when combined, can be regarded as layers of applications (see Figure 7.7). Multi process descriptions usually attempt to present various components and the high level view of inter process communications between these components. In this experiment, the scenarios primarily focused on exercising the browser components to the maximum extent with respect to typical use. It is possible to argue that typical use can mean many things. Today, browsers are used for a wide range of purposes, from browsing the web pages on WWW to watching movies. Google Chrome's salient features have been discussed in Section 6.1, one of which is advanced support for HTML5 coupled with the speed of rendering a wide range of feature rich web pages. One of the scenarios used was from the Google Chrome experiment called Arcade Fire [237], which exercises the browser by rendering a music video using the latest HTML5 features. The rendering utilises the user's local geographical map in real time, while simultaneously communicating with the remote server located on the Internet. The expectation was that IPC communication, rendering engine related features, will be exercised as a result of this scenario. The other scenario used Internet Explorer's Fish Bowl rendering speed test [238]. In addition to this, standard features like storing favourites using secure web based email were used.

Table 7.1 exhibits all the lists containing methods from features that are expected to be exercised frequently. These methods have been highlighted in red. It can also be seen that the Slim Tune profiler picked up the system's large number of built-in C++ methods related to Standard Template (STL) classes, some of which were found to have been extensively used and overridden on extended form within the Google Chrome browser. In all cases except a few, these were ignored when analysis was carried out. In all cases, untraceable methods were given the lowest score of "0", which was to ‘err on the side of caution’. This leads to a reduction of the probability density and, as a result, scores may appear to have lower performance.

In the case of experiments on JHotdraw 6.01b1 (discussed in Chapter 6) the maximum search space spanned about 71,267 lines of code, as opposed to a million lines of code in Google Chrome. The setup of Google Chrome and the manual mutation generation were conducted very conservatively and in a principled manner, and the results indicate that all the scores performed better than chance.

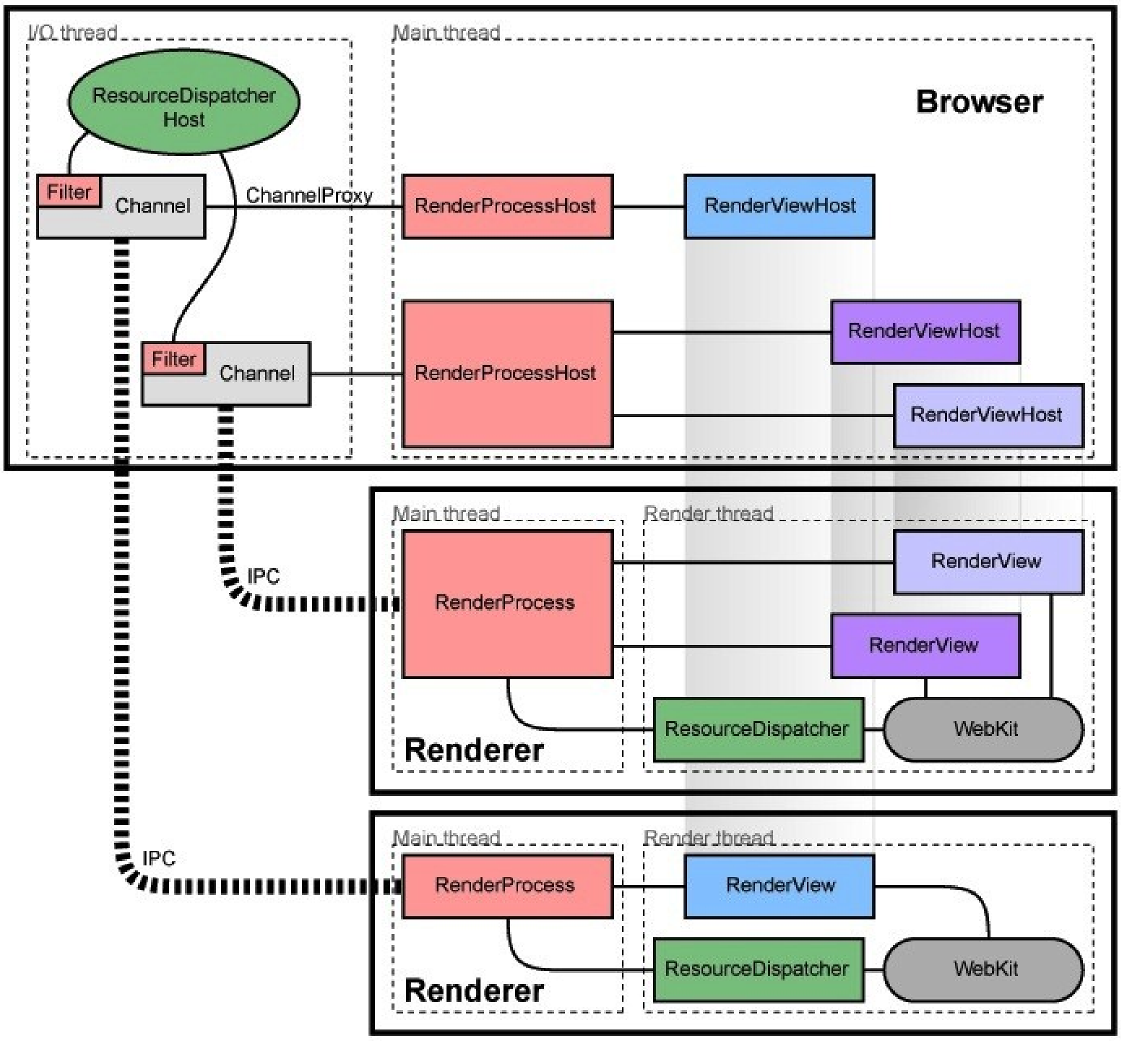
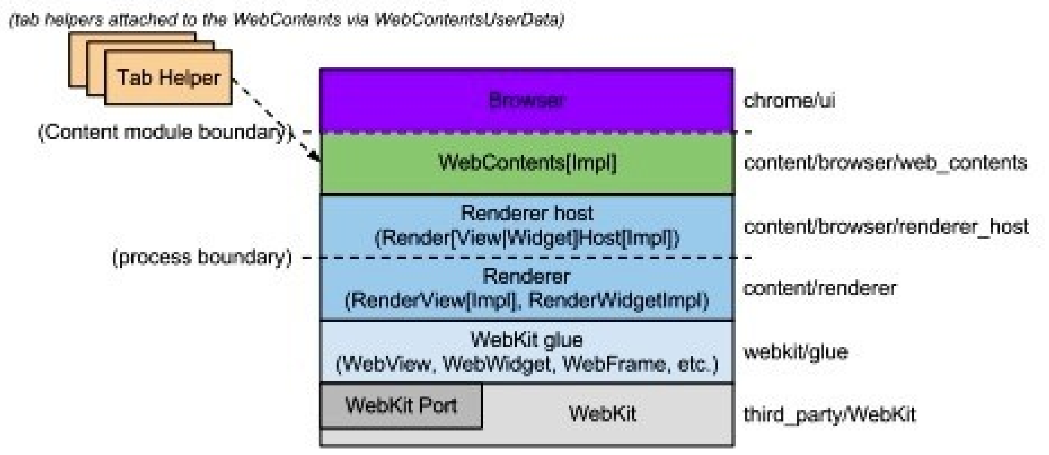


Figure 7.6 Multi Process Architecture of Google Chrome Browser [239].

Based on the concept and definition of functionally important methods (introduced in Chapter 4) results confirm that the method described in Chapter 5 can be used to predict functionally important methods with prediction performance that is better than chance.

To validate the predictions of functionally important methods, these methods were replaced by empty stubs with an appropriate input-output interface. Evaluation of the software behaviour following such alterations confirms that the predicted methods are significantly more likely to be functionally important than methods picked randomly.



(tab helpers attached to the WebContents via WebContentsUserData)
Tab Helper
(Content module boundary)
(process boundary)
Browser
WebContents[Impl]
Renderer host (Render[View|Widget]Host[Impl])
Renderer (RenderView[Impl], RenderWidgetImpl)
WebKit glue (WebView, WebWidget, WebFrame, etc.)
WebKit Port
WebKit
chrome/ui
content/browser/web_contents
content/browser/renderer_host
content/renderer
webkit/glue
third_party/WebKit

Figure 7.7 Google Chrome Conceptual Application layers [239].

A pertinent question in this context is whether the method followed could be improved, such that errors that can be attributed to the human element can be reduced in future research. Another question in the same context is whether the effort required when performing experiments using this method is the best way to validate the use of network measures in dynamic analysis of software. Some of these questions will be discussed in the concluding chapter. However, it can be stated that despite the need for further work, as proof of concept, the results obtained from this experiment provide encouraging indications of the possible uses of network analysis techniques to support program comprehension and software maintenance tasks.

Chapter 8. Discussion and Conclusion

8.1 Discussion

In Chapter 4 the following research questions were posed:

Q 1. What is functional importance in the context of software systems?

Q 2. Can network analysis be used to discover functionally important elements of object oriented software systems in the dynamic context?

Q 3. If network analysis can discover functionally important elements in object oriented software systems, how can these discoveries be verified as truly important?

Q 4. Can Cloud based testing be gainfully utilized for large scale software testing?

8.1.1 The concept of functional importance

The abstract concept of functional importance is intuitively understood and widely used in some fields of science, for instance systems biology and neuroscience. In this thesis, the concept of functional importance has been defined in the context of software engineering. A definition has been proposed based on the object-method interaction network in object oriented (OO) software systems, the functionality of OO software, and the perception of the user concerning the expected normal overall functional behaviour of the system when the system is exercised using typical user acceptance testing scenarios.

The notion of the importance of an element, or of a set of elements, for that system in the context of "what the system does" (i.e. its functionality) is not easy to define precisely. The natural questions that arise are: "how important?", "in what context is it important?", "is the degree of importance measurable in terms of its potential impact?" and "is it possible to identify these important elements with reasonable effort?"

The proposed definition is based on the notion of user perception and the expectation of normal delivery of functions of software. The use of network modelling is based on empirical scientific research evidence that shows structural-functional correlations can be modelled using scale free power law network models. Software engineers would often refer to a library as important for functionality, or would characterize part of the software as the heart of the system without which the system would fail. In the case of both large and small business critical systems, it is common practice to analyse risk and evaluate its impacts on the business should the system fail.

In dealing with large complex systems, intuition and experience alone may not be sufficient to identify correctly elements of the system that pose potential risks. Tasks of this nature are often attempted by an iterative process of 'divide and conquer'. That is, a process of progressive elimination of potential candidates that may pose a risk. This process is essentially heuristic in nature and is applicable to large and complex software.

The question arises "why dynamic analysis?" The simplest answer is that only a running system will display properties that can be experienced, which then can be evaluated against expectations of those properties. Large software today is considered sufficiently complex to be use as a experimental substitute system (e.g. in place of animal brain) on which emergent data modelling and analysis techniques are validated. The choice of complex network technique is partly motivated by the needs of the complex network research community for validation of methods proposed in this emergent field. However, the primary motivation is to explore whether the properties that scale free networks claim to have can be exploited in software engineering to gain program understanding in an efficient manner, and then to investigate whether the proposed methods can be used to obtain program information that can be used to prioritise activities like software testing. In this context, the definition of functional importance for methods of OO software proposed in this research is relevant. This definition tries to build a conceptual bridge between large object oriented software systems and scale free network models, by exploiting the properties of highly connected network hub nodes that make these networks both fragile and robust simultaneously. Research on complex network models and the structure of object oriented software has only focused on exploring whether network representations of software systems can be described by using power law and scale free network models, and not on its practical implications or application in software engineering.

8.1.2 Methodology and Results

In this thesis, a technique has been proposed to represent object-method collaboration networks built from dynamic program analysis data. Using this technique, empirical edge based network analysis measures are then used to analyse the object-method collaboration network to predict functionally important edges of the network as a ranked list of methods invoked due to scenario driven execution of the software.

The validation of the predictions of functionally important edges, and consequently the performance of the network measures, exploits the concepts of network perturbation through edge removal. A novel technique, based on mutation testing (that simulates the condition of edge removal in network graphs), has been developed and used to validate the predictions of functionally important method calls in a large OO software system.

The likelihood of a network measure producing correct prediction of functionally important methods was analysed by calculating the cumulative probability density function (a standard non-parametric statistical technique) for the results of the mutation testing. The use of the non-parametric statistical technique was to quantify the performance of these predictions and compare this against the outcomes of application of mutation testing analysis on a large set of system methods, which were not chosen using the network measures based ranking technique. The performance of mutation testing on the random set of methods was used as a benchmark for the purpose of performance comparison.

The mutation testing technique, used to verify the predictions of functional importance (obtained by application of network analysis), uses a form of mutation testing that does not rely on value replacement, but on control path removal. This mutation technique is novel because the control path removal is not a removal of the method, but of the information flow through the method, simulating edge removal in network graphs. The technique of introducing maximal mutation preserves the system in the sense that all interfaces remain unaltered, and only the method body representing an edge is short-circuited by introducing an empty stub in its place. A drawback in this state-of-the-art technique is that it currently does not account for possible multiple calling contexts.

The results presented, that underpin this thesis, are based on a set of empirical software engineering experiments. These experiments were carried out on JHotdraw6.0b1 and Google Chrome, and are the main subjects of discussion in chapters 6 and 7 respectively. Chapter 5 deals with the design of experiments and the techniques common to the discussion in chapters 6 and 7.

A summary of the results of verification is presented in Table 8.1. The table lists the connectivity distribution identified for the object-method message network representations, generated from the dynamic analysis data for both the systems. The random baseline generation found that the probability of mutation of a randomly chosen method leading to failure of the system is close to 60%. The table marks measures as "Yes" or "No" based on the performance falling above or below the random average baseline. In cases where the performance prediction is only marginally better than chance, the result has been marked as "borderline", to account for the small sample size and potential data collection tool related issues. In the case of Google Chrome, the identification of a parent class of objects was not listed, and as a result HCS(e) measure was not used. The results indicate that the Weighted Connection score (WCS(m)), the total Call Frequency score (CFS(m)), and the Betweenness score (BWS(m)) all seem to indicate better ability than chance to predict functionally important method calls in the case of Google Chrome, a software with scale free connectivity distribution. In the case of JHotdraw, where the connectivity distribution is log-linear and a small world borderline case, the indications are that BWS scoring is not a good indicator, while all other scores considered perform better than the random baseline average. In all cases the top 50 methods of all lists were evaluated. The decision to present the top 20-25 ranked positions is arbitrary. If resources and time permitted, a much larger volume of dynamic analysis data could have been analysed and it would be meaningful to choose more rank positions in the presentation. Another argument would be to present no more than the first 7-10 rank positions, taking into consideration Miller's [240] discussion on natural limits of the ability to assimilate information.

Software	**Connectivity distribution**	**Random Baseline**	**HCS(e)**	**WCS(m)**	**CFS(m)**	**CFS(p)**	**BWS(m)**
JHotdraw 6.0b1	Log-linear - Small world	0.6	Yes	Yes	Yes	Yes	No
Google Chrome	Scale Free Small world	0.59	NA	Yes	Borderline	NA	Yes

Table 8.1 Results summary of verification of functional importance predictor measures.

8.1.3 Implications

In chapters 6 and 7, the predictions obtained have been discussed in the context of the overall functionality of the two systems. This discussion happens in the circumstance where the user of these predictions is also a programmer with some appreciation of what can be expected to be happening at the code level when these systems are used. This use is in the form of typical usage. That is, if one is using a graphical application, the actions would be to draw figures and manipulate the attributes of the figures. Similarly, if one is using a web browser application, the expectation is that the user would point the application to fetch a document identified by a unique resource identifier (URL), where this document may download a static HTML document or a dynamic document that mimics the notion of interaction through responses to user actions. While deeper analysis of individual classes and methods was beyond the scope of this research, it was found that in the majority of cases the classes and methods that a programmer may find important appeared on the list. For instance, methods related to figure manipulation in Jhotdraw6.0b1, methods related to URL access, and rendering of HTML pages in Google Chrome. The disabling of these methods adversely affected normal operation of the software, and the operating system in certain cases for Google Chrome. There is an essential difference between JHotdraw and Google Chrome from the compositional point of view. JHotdraw, despite being a framework, is not designed for dynamic extension and, as a result, applications written purely with the existing interface of JHotdraw 6.0b1 are likely to be constrained for growth and unlikely to retain scale free power law properties. In contrast, Google Chrome is more recent software and is designed with modularity and dynamic extension as a goal. This coupled with the large base size makes it likely that Google Chrome has power law scale free connectivity by default as a consequence of design. In this sense, results of verification for functional importance indicate network analysis may be useful when identifying functionally important methods.

The empirical network analysis measures and the technique of mutation testing employed in the experiments are not new. The novelty of this work lies in exploitation of properties of scale free networks to define and then validate the concept of functional importance of methods of object oriented systems. Previous research along similar lines by Zaidman and Demeyer [6], to identify key classes in software, uses HITS web (data) mining algorithm to predict key classes. The key difference with our research is that the rationale behind use of the HITS algorithm based technique is not articulated. The HITS algorithm based technique applies to network graphs that are power law distributed, but the paper [6] does not verify the underlying distribution of the data. Furthermore, the paper does not propose what property of network or software makes the use of the HITS algorithm appropriate and does not verify the prediction of key classes. In previous research, Zaidman et al. [169] used the HITS technique to argue that identification of key classes can support program comprehension. Arguments along similar lines can be found in the work by Zimmermann and Nagappan [96], which uses social network analysis techniques to detect Ego Network motifs in software networks. The premise of this work [96] is that Ego networks may be indicators of locations that may be defect prone, and (therefore) are candidates that could benefit from more rigorous testing. In all these cases, the conceptual bridge between the properties of network graphs and software is tenuous at best and non-existent at worst. The concept of functional importance presented here attempts to bridge the gap that currently exists. The techniques and experimental results presented in this thesis adopt a more rigorous ground up approach to verify that network analysis measures can truly identify functionally important elements of a software system, and thereby the key classes in object oriented software.

The review of literature highlighted that the challenges in static and dynamic analysis are primarily the challenges of dealing with large volumes of program data. The experience of this research has not been different, where challenges in collection of data have been largely overcome, but efficient and scalable data processing has only partially been addressed.

In [14], discussion on use of the Cloud for large scale testing has demonstrated that the Cloud can be considered a viable platform for software testing. The workflow used to test the Cloud, and the techniques developed to launch and manage the platform, processes and the data, is sufficiently generic to be adopted for similar endeavours. The challenges of large volume data management and computation faced in this research can be considered analogous to similar challenges in industrial environments, particularly for software testing.

The use of Cloud based testing in the context of this research was primarily to address challenges of large volume mutant generation in a limited time span. Despite various limitations, use of the Cloud was found to be viable and potentially beneficial for software testing, similar to the mutation testing done in the experiments on Google Chrome. The main attractions behind use of the Cloud for this experiment were, the relatively low cost of unit usage, the flexibility to configure to very specific technical requirements (e.g. facility to choose a particular type of processor and the machine contention factor), and the on demand availability of resources. The experience during the latter part of this work has been that the Cloud is as efficient as the most inefficient software used on it. It has been noticed that inefficient software infrastructure or workflow in the Cloud can be very expensive, relative to captive computation resources.

8.1.4 Limitations

Validation Technique

The technique used to perform validation mimics network perturbation through edge removal. The aim is to verify whether such removal leads to network fragmentation and impairment of the functional network.

The technique, when applied to software systems, is the most up to date. However, it still needs refinement. In a modern multi-threaded object oriented program, a method body may not have only one execution context; for instance it can be using another class as a field that may be implementing the observer pattern and executing on a different thread. The mutation of the method body can mean disabling multiple control paths. The method of mutant generation by using maximalist stub needs to be further refined by considering how multiple threads of execution can be accounted for and handled in the stub. Refinements are likely to lead to a more robust validation of the functional importance of methods proposed in this research. The technique is extremely expensive in terms of human effort, and obviously prone to human error due to the long attention span required to ensure that each manual mutation is correctly implemented and recorded. However, to the best of our knowledge, this is the best technique currently available to simulate perturbation of a network.

Data Generation

In the series of experiments conducted in this research, the generation of dynamic analysis data has been a major challenge. This is primarily due to the fragility of the tools, which were primarily those from the Eclipse TPTP project in the case of software written in Java. In the case of software written in C++, the problem has been in the format and quality of data generated by the tools. A general observation, applicable to existing tools in the context of their capability to handle large pieces of software, is fragility, especially when dealing with execution trace generation from typical usage scenarios.

Most experiments tend to execute short scenarios on relatively small scale open source software. In contrast, our research data was generated for fairly large and popular open source software by executing typical usage scenarios. This resulted in execution trace files of size ranging from 500 Megabytes (MB) to 120 Gigabytes (GB) each. In the case of TPTP XML4Profiling format files, the standard technique to process these files is to use a SAX event based XML parser. These tools have been produced as part of this research, and a Cloud enabled processing pipeline has been implemented. The problem of scalability remains, primarily because the production quality libraries used have not been engineered to handle such large volumes of data in individual files.

Network representation

In our experiments, undirected network representation has been used. In itself, this is not a problem, because the primary issue explored in this research is whether network analysis can identify important elements of the functional network, rather than the control flow within the functional network.

The problem is that most network analysis techniques ignore loops or cycles in the network representation. In software systems, not considering cycles in the functional network and not considering subgraph branches that may represent concurrent processing (i.e. observer patterns) creates the problem of discarding runtime information that may be important and may contribute to the scores.

In the literature, discussion of this aspect of complex network modelling is limited, particularly in the context of software systems, which gives the impression that the absence of research is due to pragmatic reasons, considering the difficulty in generating multi labelled graphs accurately.

Statistical analysis

This research has a strong element of experimental software engineering, and while the results are encouraging, the volume of experiments that was possible was limited by the validation technique adopted. This limitation, coupled with several limitations in the data collection infrastructure, reduced the scope of producing more statistically significant results. The statistical technique adopted is standard, but the sample size needs to be significantly increased by extending these experiments and considering new systems. Efforts in this respect have already resulted in the collection of a large volume of dynamic analysis trace data. However, the question now is how to automate the process of mutation generation in order to improve process efficiency.

8.2 Conclusion

This thesis proposes the concept and definition of functional importance for the methods of object oriented software. This research tests the hypothesis that elements of an object oriented system that are functionally important can be detected by using network analysis, in conjunction with scenario driven dynamic analysis. The thesis approaches the analysis of dynamic method-object interaction by using network representation, and linking the representation and properties of the network model to the properties in the software structural-functional space. This ground up approach and subsequent validation of the hypothesis represents a significant deviation from previous research, which used network models to reason about the structure and behaviour of software systems.

A definition of functional importance has been given, and network analysis has been used to predict the functionally important methods in object oriented software. Mutation testing has been used to simulate network edge removal and network fragmentation. This testing technique has been used to validate the idea that the network analysis measures are truly capable of identifying the functionally important structural elements of the dynamic software network. It is possible to see the application of this work in program reverse engineering, program comprehension, software test prioritisation, program vulnerability analysis, and other activities that deal with post release software maintenance.

This research asked four questions. In response, on the basis of the work presented, it is possible to claim that: a) our definition of functional importance captures the essence of this abstract concept in the context of the current knowledge of object oriented software systems; b) network analysis can be used to discover functionally important method calls; c) the verification of predicted important method calls should be considered as significant, although the initial results are not sufficient and the process needs more empirical validation; and d) the Cloud can be gainfully utilized for software testing, although the use of the platform should be guided by significant gains in testing process efficiency.

At this point it is relevant to ask about the likelihood that this work will have immediate applicability in software engineering practice. It is not uncommon in software engineering that an early metric definition is adopted by the community, while later refinements fail to gain popularity. An example is C&K metrics and Briand's refinements of the C&K metrics. The use of complex network modelling in software engineering is in its infancy, where the only techniques that seem to have gained some acceptance are based on the use of social networks analysis (SNA).The impact of application for this family of models and techniques, to analyse a software program's static or dynamic data, cannot be determined from the limited number of available literature. In summary, it is too early to know for sure.

In contrast to previous work involving the use of complex network models in software engineering and program analysis, this research has made a conscious effort to define a concept for software systems, and to underpin this definition in terms of a well-known property of scale free small world complex network models. This property deals with the robust but fragile nature of important structural nodes of such networks. The application of network analysis has been motivated by the aim to identify these nodes in software networks and validate these predictions experimentally. This validation forms the first step towards further empirical validation. Thus, although the validation could be stronger with better tool support, it has so far produced encouraging results. This provides further opportunity to work on a more rigorous definition and perform further experimental validation of the network metrics.

8.3 Future work

This work presented several practical challenges, which have been discussed in Section 8.1. Continuation of this research is motivated primarily by these challenges and the possible solutions that may improve the quality of the results and the efficiency of the research cycle (i.e. data collection, data analysis and interpretation).

8.3.1 Experiment cycle

Validation technique

The manual validation technique used in this work was extremely resource and labour intensive and prone to human error. It is known that dynamic instrumentation tools provide the necessary facilities to analyse the function names and instructions from the binary code, if the code is generated using the necessary debug flags.

It may be possible to collect traces that are annotated with control path information and also with what led to that path being executed. If the predictions are generated based on these traces, all the context information is likely to be present that may eliminate the need to generate mutants. In fact mutations may be dynamic, and the methods and annotations may form the necessary data to perform context aware mutations at runtime.

Such a scheme will require that the software subjected to the dynamic mutation test is executed on a virtual machine, so that fatal mutations do not affect the host environment. The good news is that both Java's virtual Machine and Pin Tools' virtual machine provides the necessary isolation. In this context, as part of the evaluation, a tool was written to investigate whether mutation testing could be automated. This tool functions as expected, but it requires further work. At this point the tool relies on standard method names, which is acceptable in Java. However, for C++ it fails to distinguish method names, which is not mangled using a standard scheme across compilers.

This type of context aware mutation is a refinement that can benefit the experiment process by making it more robust and efficient.

Data analysis

The network data analysis tools in this project were written as GUI driven tools, which needs RIGI format files as input. These RIGI format files are not strictly necessary, and network data generation should be a one step process from the trace file. A tool written in C++ has been developed for this purpose. This tool can ingest a trace in standard format and generate two representations of the network (directed and undirected). It is also able to compute the network analysis measures. The advantage of this tool is that it works both as a GUI and in console mode, making it possible to use in batch mode remotely in the Cloud. This tool needs to be tested and the numerical results validated before it can be used in a Cloud based analysis chain. The primary advantage of this type of tool is the ability to carry out complimentary forms of analysis using the same data set in an automated way, thereby reducing the need for human involvement except at the start and end of the process.

An important aspect of the data analysis is the ability to parse efficiently very large trace data files. In this context, one technique that can be explored is trace abstraction, presented in [180] and in earlier work by the same author.

In addition to this, it is worth exploring whether the NoSQL Database and software like Hadoop could be used in the Cloud to mitigate the challenges posed to efficient large trace file processing.

8.3.2 Extension of empirical experiments

In chapters 5, 6 and 7 it was mentioned that the results would cover only part of the data that was collected as part of this work. A significant volume of new data, totalling approximately 1.5 terabytes, is now available for further validation work and extension of the empirical experiments.

At this point the work only claims the ability to identify functionally important methods in the software. However, it is more meaningful to develop the ability to analyze the degree of importance of a functionally important method. In this context, it may be worth exploring whether concepts such as design pattern extraction and stereotype extraction, which use static analysis techniques, can be utilized to associate contextual information in the prediction of importance, based on network analysis of the traces.

The work presented here is motivated in light of the limitations of static analysis. A logical extension would be to compare the network analysis-based prediction of functionally important methods with a popular static analysis-based approaches. This could also lead to production of a gold set against which performance of network analysis methods is compared as opposed to the baseline used in this work.

Bibliography

[1] BBC," *RBS software failure.*", Available:http://www.bbc.co.uk/news/business-18575932, Last Accessed on 16/02/2013,2012.

[2] NIST and RTI, "The economic impacts of inadequate infrastructure for software testing.", National Institute of Standards and Technology RTI Project Number 7007.011, 2002.

[3] G. Goth, "Ultralarge Systems: Redefining Software Engineering?", IEEE *Software,* vol. 25, pp. 91-94, 2008.

[4] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks", *Reviews of Modern Physics,* vol. 74, pp. 48-94, 2002.

[5] M. Newman*, et al.*,"*The Structure and Dynamics of Networks*", Princeton University Press, 2006.

[6] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques", *J. Softw. Maint. Evol.,* vol. 20, pp. 387-417, 2008.

[7] A.-L. Barabási and E. Bonabeau, "Scale-free networks", *Scientific American,* vol. 288, pp. 50-59, 2003.

[8] H. Jeong*, et al.*, "The large-scale organization of metabolic networks", *Nature,* vol. 407, pp. 651 - 654, 2000.

[9] E. R. Kandel*, et al.*, *Principles of Neural Science*, 4th ed. New York: McGraw-Hill, 2000.

[10] A. Pakhira and P. Andras, "Can we use network analysis methods to discover functionally important method calls in software systems by considering dynamic analysis data?", Proceedings of WCRE-PCODA,pp. 12-19, 2010.

[11] A. Pakhira and P. Andras, "Dynamic network analysis of software systems", Newcastle University, CS-TR-1240, 2011.

[12] A. Pakhira and P. Andras, "Validation of Network Analysis Methods Applied in the Context of Dynamic Analysis of Software Systems", Newcastle University CS-TR-1239, 2011.

[13] A. Pakhira and P. Andras, "Using Network Analysis Metrics to Discover Functionally Important Methods in Large-Scale Software Systems", Proceedings of ICSE-*WETSoM,* pp.70-76 , 2012.

[14] A. Pakhira and P. Andras, "Leveraging the Cloud for Large-Scale Software Testing – A Case Study: Google Chrome on Amazon", *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, ed: IGI Global,Chapter. 12, pp. 252-279, 2013.

[15] P. Andras*, et al.*, "A Measure to Assess the Behavior of Method Stereotypes in Object-Oriented Software", Proceedings of *ICSE-WETSoM*, pp. 7-13, 2013.

[16] B. W. Boehm, "Software Engineering Economics", IEEE Transactions on Software Engineering*,* vol. SE-10, pp. 4-21, 1984.

[17] W. W. Royce, "Managing the development of large software systems: concepts and techniques", Proceedings of IEEE WESTCON, pp. 328-338, 1970.

[18] H. Mooz and K. Forsberg, "System Engineering for Faster, Cheaper, Better", *Center for Systems Management,* pp. 1-11, 1998.

[19] M. M. Lehman, "Programs, life cycles, and laws of software evolution", *Proceedings of the IEEE,* vol. 68, pp. 1060-1076, 1980.

[20] T. Eisenbarth*, et al.*, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", Proceedings of ICSM, pp. 602-611, 2001.

[21] R. Petrasch, "The Definition of ‚Software Quality: A Practical Approach", Proceedings of *ISSRE*, pp. 33-34, 1999.

[22] L. A. Laranjeira, "Software size estimation of object-oriented systems", *IEEE Transactions on Software Engineering, ,* vol. 16, pp. 510-522, 1990.

[23] R. E. Park, "Software Size Measurement: A Framework for Counting Source Statements", Carnegie Mellon University, CMU/SEI-92-TR-020, 1992.

[24] H. B. K. Tan*, et al.*, "Conceptual data model-based software size estimation for information systems", *ACM Trans. Softw. Eng. Methodol.,* vol. 19, pp. 1-37, 2009.

[25] A. Zeller, "*Why Programs Fail,A Guide to Systematic Debugging",* 2ed.: MORGAN KAUFMANN, 2009.

[26] D. Talby*, et al.*, "Agile software testing in a large-scale project", *IEEE Software,* vol. 23, pp. 30-37, 2006.

[27] D. Gelperin and B. Hetzel, "The growth of software testing", *Commun. ACM,* vol. 31, pp. 687-695, 1988.

[28] D. Jackson, "A direct path to dependable software", *Commun. ACM,* vol. 52, pp. 78-88, 2009.

[29] N. Fenton and S. L. Pfleeger, "*Software Metrics A rigorous and practical approach"*, 2 ed. Massachusetts,USA: PWS Publishing Co., 1997.

[30] T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, vol. SE-2, pp. 308-320, 1976.

[31] B. G. Ryder, "Constructing the Call Graph of a Program", *IEEE Transactions on Software Engineering,* vol. SE-5, pp. 216-226, 1979.

[32] E. Yourdon and L. L. Constantine, "*Structured Design", Fundamentals of a Discipline of Computer Program and Systems Design*: Prentice-Hall, Inc., 1979.

[33] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering,* vol. SE-7, pp. 510-518, 1981.

[34] J. Kearney, P.*, et al.*, "Software complexity measurement", *Commun. ACM,* vol. 29, pp. 1044-1050, 1986.

[35] E. Arisholm*, et al.*, "Dynamic Coupling Measurement for Object-Oriented Software", *IEEE Trans. Softw. Eng.,* vol. 30, pp. 491-506, 2004.

[36] M. D. Ernst, "Static and dynamic analysis: synergy and duality", Proceedings of ICSE-WODA, pp.24-27 , 2003.

[37] A. Von Mayrhauser and A. Marie Vans, "Program comprehension during software maintenance and evolution", *Computer,* vol. 28, pp. 44-55, 1995.

[38] C. McMillan*, et al.*, "Combining textual and structural analysis of software artifacts for traceability link recovery", Proceedings of ICSE-TEFSE, pp. 41-48, 2009.

[39] G. Ammons*, et al.*, "Mining specifications", *SIGPLAN Not.,* vol. 37, pp. 4-16, 2002.

[40] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Trans. Softw. Eng.,* vol. 20, pp. 476-493, 1994.

[41] L. C. Briand*, et al.*, "A unified framework for cohesion measurement in object-oriented systems", Proceedings of *METRICS*, pp. 43-53, 1997.

[42] L. C. Briand*, et al.*, "A Unified Framework for Coupling Measurement in Object-Oriented Systems", *IEEE Trans. Softw. Eng.,* vol. 25, pp. 91-121, 1999.

[43] M. H. Halstead, "*Elements of Software Science", (Operating and programming systems series)*: Elsevier Science Inc., 1977.

[44] L. D. Misek-Falkoff, "A unification of Halstead's Software Science counting rules for programs and English text, and a claim space approach to extensions", *SIGMETRICS Perform. Eval. Rev.,* vol. 11, pp. 80-114, 1982.

[45] M. E. Fagan, "Advances in Software Inspections", *IEEE Trans. Softw. Eng.,* vol. 12, pp. 744-751, 1986.
[46] M. E. Fagan, "Design and code inspections to reduce errors in program development", *IBM Syst. J.,* vol. 38, pp. 258-287, 1999.
[47] A. A. Porter*, et al.*, "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development", *IEEE Trans. Softw. Eng.,* vol. 23, pp. 329-346, 1997.
[48] T. Kamiya*, et al.*, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering,* vol. 28, pp. 654-670, 2002.
[49] H. Ogasawara*, et al.*, "Experiences with program static analysis", Proceedings of *METRICS*, pp. 109-112, 1998.
[50] A. Nathaniel*, et al.*, "Evaluating static analysis defect warnings on production software",Proceedings of PASTE, pp.1-8, 2007.
[51] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis" , *Inf. Softw. Technol.,* vol. 53, pp. 363-387, 2011.
[52] P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs", *Proceedings of the second international symposium on Programming*, pp. 106-130, 1976.
[53] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", Proceedings of POPL, pp. 238-252, 1977.
[54] P. Cousot, "*Semantic foundations of program analysis",* Program Flow Analysis: Theory and Applications,S.S. Muchnick and N.D. Jones, editors,Chapter 10, pp. 303-342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1981.
[55] P. Cousot, "Program analysis: the abstract interpretation perspective", *ACM Comput. Surv.,* vol. 28, p. 165, 1996.
[56] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools", *Electron. Notes Theor. Comput. Sci.,* vol. 217, pp. 5-21, 2008.
[57] D. Challet and A. Lombardoni, "Bug propagation and debugging in asymmetric software structures", *Physical Review E,* vol. 70, p. 046109, 2004.
[58] A. E. Hassan and R. C. Holt, "The top ten list: dynamic fault prediction", Proceedings of *ICSM*, pp. 263-272, 2005.
[59] L. Briand*, et al.*, "Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Directions", *Empirical Software Engineering,* vol. 4, pp. 387-404, 1999.
[60] L. C. Briand and J. Wust, "Empirical Studies of Quality Models in Object-Oriented Systems", *Advances in Computers,* vol. 56, pp. 98-167, 2002.
[61] Y. Shin*, et al.*, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities", *IEEE Transactions on Software Engineering ,* vol. 37, pp. 772-787, 2011.
[62] R. C. Martin, "Design Principles and Design Patterns", *www.objectmentor.com,* p. 34, 2000.
[63] C. Zhang and H.-A. Jacobsen, "Efficiently mining crosscutting concerns through random walks", Proceedings of AOSD, pp.226-238, 2007.
[64] S. Jenkins and S. R. Kirk, "Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution", *Inf. Sci.,* vol. 177, pp. 2587-2601, 2007.
[65] B. Dit*, et al.*, "Feature location in source code: a taxonomy and survey", *Journal of Software: Evolution and Process,*vol.25 , pp. 53-95, 2013.

[66] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension", Proceedings of IWPC, pp. 271-278, 2002.

[67] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph", Proceedings of IWPC, pp.241-249, 2000.

[68] Y. Liu, "Modeling class cohesion as mixtures of latent topics", Proceedings of *ICSM*, pp. 233-242, 2009.

[69] E. Hill*, et al.*, "Automatically capturing source code context of NL-queries for software maintenance and reuse", Proceedings of ICSE, pp.232-242, 2009.

[70] E. Gamma*, et al.*, "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley Professional, 1994.

[71] G. El Boussaidi and H. Mili, "Understanding design patterns — what is the problem?", *Software: Practice and Experience,* pp. 1495-1529, 2012.

[72] C. Zhang and D. Budgen, "What Do We Know About the Effectiveness of Software Design Patterns", *IEEE Transactions on Software Engineering,* vol.38, pp. 1213 - 1231, 2012.

[73] N. Dragan*, et al.*, "Reverse Engineering Method Stereotypes", Proceedings of *ICSM*, pp. 24-34, 2006.

[74] N. Dragan*, et al.*, "Using method stereotype distribution as a signature descriptor for software systems", Proceedings of *ICSM*, pp. 567-570, 2009.

[75] N. Dragan*, et al.*, "Automatic identification of class stereotypes", Proceedings of *ICSM*, pp. 1-10, 2010.

[76] Y.-G. Guéhéneuc*, et al.*, "Improving design-pattern identification: a new approach and an exploratory study", *Software Quality Journal,* vol. 18, pp. 145-174, 2010.

[77] G. El Boussaidi and H. Mili, "Understanding design patterns — what is the problem?", *Software: Practice and Experience,* vol. 42, pp. 1495-1529, 2012.

[78] R. Diestel, "*Graph Theory"*, 3 ed. vol. 173. Heidelberg,New York: Springer-Verlag,p. 415 , 2005.

[79] P. Erdős and A. Rényi, "On the strength of connectedness of a random graph", *Acta Mathematica Hungarica,* vol. 12, pp. 261-267, 1961.

[80] M. E. J. Newman*, et al.*, "Random graph models of social networks", *Proceedings of the National Academy of Sciences of the United States of America,* vol. 99, pp. 2566-2572, 2002.

[81] E. Bullmore and O. Sporns, "Complex brain networks: graph theoretical analysis of structural and functional systems", *Nat Rev Neurosci,* vol. 10, pp. 312-312, 2009.

[82] Central Limit Theorem. Available: http://www.math.uah.edu/stat/sample/CLT.html, Last Accessed on 16/02/2013.

[83] J. Travers and S. Milgram, "An Experimental Study of the Small World Problem", *Sociometry,* vol. 32, pp. 425-443, 1969.

[84] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks", *Nature,* vol. 393, pp. 440 - 442, 1998.

[85] L. A. Amaral*, et al.*, "Classes of small-world networks", *Proc Natl Acad Sci USA,* vol. 97, pp. 11149 - 11152, 2000.

[86] A. L. Barabasi and R. Albert, "Emergence of scaling in random networks", *Science,* vol. 286, pp. 509 - 512, 1999.

[87] R. Albert, "Scale-free networks in cell biology", *J Cell Sci,* vol. 118, pp. 4947 - 4957, 2005.

[88] A.-L. Barabási, "Scale-Free Networks: A Decade and Beyond", *Science,* vol. 325, pp. 412-413, 2009.

[89] L. d. F. Costa*, et al.*, "Characterization of complex networks: A survey of measurements", *Advances In Physics,* vol. 56, pp. 1-84, 2007.

[90] L. Li*, et al.*, "Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications (Extended Version)", *Internet Mathematics,* vol. 2, pp. 431-523, 2005.

[91] A. Clauset*, et al.*, "Power-Law Distributions in Empirical Data", *SIAM REVIEW,* vol. 51, pp. 661-703, 2009.

[92] S. N. Dorogovtsev and J. F. F. Mendes," *Evolution of Networks: From Biological Nets to the Internet and WWW (Physics)"*, Oxford University Press, Inc., p.280, 2003.

[93] M. E. Newman, "Assortative mixing in networks", *Phys Rev Lett,* vol. 89, p. 208701, 2002.

[94] M. E. Newman, "The structure and function of complex networks", *SIAM REVIEW,* vol. 45, pp. 167 - 256, 2003.

[95] R. Milo*, et al.*, "Network Motifs: Simple Building Blocks of Complex Networks", *Science,* vol. 298, pp. 824-827, 2002.

[96] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs", Proceedings of ICSE, pp. 531-540, 2008.

[97] D. Koschützki*, et al.*, "Centrality Indices", *Network Analysis*. vol. 3418, U. Brandes and T. Erlebach, Eds., ed: Springer Berlin Heidelberg, pp. 16-61, 2005.

[98] M. E. Newman, "Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality", *Phys Rev E Stat Nonlin Soft Matter Phys,* vol. 64, pp. 016132-1-016132-7, 2001.

[99] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness", *Sociometry,* vol. 40, pp. 35-41, 1977.

[100] M. E. J. Newman, "Modularity and community structure in networks", *proceedings of National Academy of Sciences,* vol. 103, pp. 8577–8582, 2006.

[101] J. Yoon*, et al.*, "An algorithm for modularity analysis of directed and weighted biological networks based on edge-betweenness centrality", *Bioinformatics,* vol. 22, pp. 3106-3108, 2006.

[102] M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law", *Contemporary Physics,* vol. 46, pp. 323-351,2005.

[103] M. E. J. Newman, "Mixing patterns in networks", *Physical Review E,* vol. 67, p. 026126, 2003.

[104] M. E. Newman, "Scientific collaboration networks. I. Network construction and fundamental results", *Phys Rev E Stat Nonlin Soft Matter Phys,* vol. 64, pp. 016131-1-016131-8, 2001.

[105] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks", *Proceedings of the National Academy of Sciences (PNAS),* vol. 99, pp. 7821-7826, 2002.

[106] A. Tosun*, et al.*, "Validation of network measures as indicators of defective modules in software systems", Proceedings of PROMISE,Article No. 5, 2009.

[107] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks", *Physical Review E,* vol. 69, p. 026113, 2004.

[108] E. Almaas, "Biological impacts and context of network theory", *J Exp Biol,* vol. 210, pp. 1548-1558, 2007.

[109] S. Valverde*, et al.*, "Scale-free networks from optimal design", *EPL (Europhysics Letters),* vol. 60, pp. 512–517, 2002.

[110] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs", *Physical Review E,* vol. 68, pp. 046116-1-046116-15, 2003.

[111] G. Concas*, et al.*, "Power-Laws in a Large Object-Oriented Software System", *IEEE Transactions on Software Engineering,* vol. 33, pp. 687-708, 2007.

[112] A. Murgia, "Time evolution and distribution analysis of software bugs from a complex network perspective", PhD Thesis, Dept. of Electrical and Electronic Engineering, University of Cagliari, 2011.

[113] R. Wirfs-Brock*, et al.*, "*Designing object-oriented software*", Prentice-Hall, Inc., 1990.

[114] R. Wheeldon and S. Counsell, "Power Law Distributions in Class Relationships", Proceedings of *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 1-10, 2003.

[115] X. Zheng*, et al.*, "Analyzing open-source software systems as complex networks", *Physica A: Statistical Mechanics and its Applications,* vol. 387, pp. 6190-6200, 2008.

[116] L. Šubelj and M. Bajec, "Software systems through complex networks science: review, analysis and applications", Proceedings of the First International Workshop on Software Mining,pp. 9-16 , 2012.

[117] I. Terekhov, "Meta-computing at D0", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment,* vol. 502, pp. 402-406, 2003.

[118] C. Severance and K. Dowd, "*High Performance Computing"*, Second Edition ed.: O'Reilly Media, 1998.

[119] S. Chaudhry*, et al.*, "High-performance throughput computing", *IEEE Micro ,* vol. 25, pp. 32-45, 2005

[120] I. Foster and C. Kesselman, "*The Grid 2: Blueprint for a New Computing Infrastructure*", Morgan Kaufmann Publishers Inc., 2003

[121] Globus.org. *Globus*. Available: http://www.globus.org/, *Last Accessed on 16/02/2013.*

[122] UKeScience. *National Grid Service(NGS)*. Available: http://www.ngs.ac.uk/, Last Accessed on 16/02/2013.

[123] US. NSF. *TeraGrid*. Available: https://www.teragrid.org/, Last Accessed on 16/02/2013

[124] S. Nanda and T.Chiueh, "A Survey on Virtualization Technologies", State University of New York, p. 42, 2005.

[125] P. Barham*, et al.*, "Xen and the art of virtualization" Proceedings of SOSP, Bolton Landing, NY, USA, 2003.

[126] E. Pagden, "The IT Utility Model—Part I", Sun Professional Services, 2003.

[127] E. Pagden, "The IT Utility Model—Part II", Sun Professional Services, 2003.

[128] L. M. Riungu*, et al.*, "Software Testing in the Cloud", Proceedings of *2nd International Workshop on Software Testing in the Cloud Co-located with the 3rd IEEE International Conference on Software Testing, Verification, and Validation (ICST 2010)*, 2010

[129] Green Hat. Available: http://www.greenhat.com/ , Last Accessed on 16/02/2013.

[130] A. M. Turing, "On Computable Numbers, with an application to the Entscheidungsproblem", *Proc. London Math. Soc.,* vol. 2, pp. 230-265, 1936.

[131] F. C. Williams and T. Kilburn, "Electronic Digital Computers", *Nature,* vol. 4117, pp. 487-487, 1948.

[132] A. Abran*, et al.*, Eds., "*SWEBOK Guide to the Software Engineering Body of Knowledge V3",* Pages. 202, 2004.

[133] A. Glenn*, et al.*, "Exploiting hardware performance counters with flow and context sensitive profiling", *SIGPLAN Not.,* vol. 32, pp. 85-96, 1997.

[134] T. Ball, "The concept of dynamic analysis", Proceedings of ESEC/FSE, pp. 216-234, 1999.

[135] M. M. Lehman*, et al.*, "Metrics and laws of software evolution-the nineties view", Proceedings of *METRICS*, pp. 20-32, 1997.

[136] R. Milner, "*Communicating and Mobile Systems: the Pi-Calculus*", Cambridge University Press, 1999.

[137] W. Fokkink, "*Introduction to Process Algebra",* vol. 8, Springer, 2000.

[138] P. Cousot, "Abstract Interpretation: Achievements and Perspectives", Proceedings of *SSGRR*, pp. 1-7, 2000.

[139] W. E. Howden, "Functional Program Testing," *Software Engineering, IEEE Transactions on,* vol. SE-6, pp. 162-169, 1980.

[140] W. E. Howden, "The Theory and Practice of Functional Testing", *IEEE Softw.,* vol. 2, pp. 6-17, 1985.

[141] J. Yue and M. Harman, "An Analysis and Survey of the Development of Mutation Testing", *IEEE Transactions on Software Engineering,* vol. 37, pp. 649-678, 2011.

[142] B. Cornelissen*, et al.*, "Execution trace analysis through massive sequence and circular bundle views", *J. Syst. Softw.,* vol. 81, pp. 2252-2268, 2008.

[143] S. Voigt*, et al.*, "Object aware execution trace exploration", Proceedings of *ICSM*, pp. 201-210, 2009.

[144] B. Meyer, "Seven Principles of Software Testing", *IEEE Transactions on Computer,* vol. 41, pp. 99-101, 2008.

[145] J. Voas*, et al.*, "Predicting where faults can hide from testing", *IEEE Software,* vol. 8, pp. 41-48, 1991.

[146] J. M. Voas and K. W. Miller, "Software testability: the new verification", *IEEE Software,* vol. 12, pp. 17-28, 1995.

[147] J. Voas and J. Payne, "Dependability certification of software components", *Journal of Systems and Software,* vol. 52, pp. 165-172, 2000.

[148] D. Jackson and M. Rinard, "Software analysis: a roadmap", Proceedings of FOSE, pp. 133-145, 2000.

[149] D. G. Waddington*, et al.*, "Dynamic Analysis and Profiling of Multithreaded Systems", *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, Y. Wiseman and S. Jiang, Eds., ed: IGI Global, pp. 156-199 , 2009.

[150] Geimer*, et al.*, "A Generic and Configurable Source-Code Instrumentation Component",Proceedings of ICCS, pp. 696-705, 2009.

[151] H. W. Cain*, et al.*, "A Callgraph-Based Search Strategy for Automated Performance Diagnosis (Distinguished Paper)", Proceedings of the Euro-Par, pp. 203-217, 2000.

[152] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation",Proceedings of PLDI,pp.89-100, 2007.

[153] R. Hundt, "HP Caliper: a framework for performance analysis tools", *IEEE Concurrency, ,* vol. 8, pp. 64-71, 2000.

[154] B. Moshe. ,"Analyzing Parallel Programs with Pin",IEEE Computer, pp. 34-41, 2010.

[155] A. Skaletsky*, et al.*, "Dynamic Program Analysis of Microsoft Windows Applications", Proceedings of *International Symposium on Performance Analysis of Software and Systems*, pp. 2-12, 2010.

[156] Intel. *Pin Tool*. Available: http://www.pintool.org/, Last Accessed on 16/02/2013.

[157] VMWARE and MIT. *DynamoRIO*. Available: http://dynamorio.org/,Last Accessed on 16/02/2013.

[158] W. Binder*, et al.*, "Advanced Java bytecode instrumentation", Proceedings of PPPJ, pp. 135-144, 2007.

[159] E. Tanter*, et al.*, "Composition of dynamic analysis aspects", Proceedings of GPCE, Eindhoven, pp.113-122, 2010.

[160] R. Laddad, "*AspectJ in Action: Enterprise AOP with Spring Applications*", Manning Publications Co., 2009.

[161] IBM. *aspectj*. Available: http://eclipse.org/aspectj/,Last Accessed on 16/02/2013.

[162] M. D. M. Couture*, et al.*, "Monitoring and tracing of critical software systems State of the work and project definition," *DRDC Valcartier TM 2008-144,Technical Memorandum,* 2008.

[163] Eclipse TPTP ProbeKit. Available: http://www.eclipse.org/tptp/platform/documents/probekit/probekit.html, Last Accessed on 16/02/2013.

[164] Eclipse TPTP Tracing and profiling tool. Available: http://www.eclipse.org/projects/project.php?id=tptp.performance, Last Accessed on 16/02/2013.

[165] Q. Zhao*, et al.*, "How to do a million watchpoints: efficient debugging using dynamic instrumentation", Proceedings of CC-ETAPS, pp. 147-162 , 2008.

[166] Gcc-instrumentation. Available: http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Code-Gen-Options.html, Last Accessed on 16/02/2013.

[167] GNU Compiler Collection. *GCC*. Available: http://gcc.gnu.org/, Last Accessed on 16/02/2013.

[168] J. K. Chhabra and V. Gupta, "A Survey of Dynamic Software Metrics", *Journal of Computer Science and Technology,* vol. 25, pp. 1016-1029, 2010.

[169] A. Zaidman*, et al.*, "Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process", Proceedings of CSMR, pp. 134-142, 2005.

[170] N. Duhan*, et al.*, "Page Ranking Algorithms: A Survey", Proceedings of *IEEE International Advance Computing Conference*, pp. 1530-1537, , 2009.

[171] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Comput. Netw. ISDN Syst.,* vol. 30, pp. 107-117, 1998.

[172] A. Potanin*, et al.*, "Scale-free geometry in OO programs", *Commun. ACM,* vol. 48, pp. 99-103, 2005.

[173] B. Cornelissen*, et al.*, "A Systematic Survey of Program Comprehension through Dynamic Analysis", *IEEE Transactions on Software Engineering ,* vol. 35, pp. 684-702, 2009.

[174] B. Cornelissen, "Evaluating Dynamic Analysis Techniques for Program Comprehension", PhD Thesis, Institute for Programming research and Algorithmics, TUDelft, Amsterdam, 2009.

[175] B. Shneiderman, "*Designing the User Interface: Strategies for Effective Human-Computer Interaction*", Addison-Wesley Longman Publishing Co., Inc., 1997.

[176] A. Lienhard*, et al.*, "Taking an object-centric view on dynamic information with object flow analysis," *Comput. Lang. Syst. Struct.,* vol. 35, pp. 63-79, 2009.

[177] J. Bohnet*, et al.*, "Visualizing massively pruned execution traces to facilitate trace exploration", Proceedings of *VISSOFT*, pp. 57-64, 2009.

[178] J. Singer and C. Kirkham, "Dynamic analysis of Java program concepts for visualization and profiling", *Sci. Comput. Program.,* vol. 70, pp. 111-126, 2008.

[179] A. Hamou-Lhadj*, et al.*, "Recovering Behavioral Design Models from Execution Traces", Proceedings of CSMR, pp. 112-121, 2005.

[180] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", Proceedings of ICPC, pp. 181-190, 2006.

[181] A. Kuhn and O. Greevy, "Exploiting the Analogy Between Traces and Signal Processing", Proceedings of *ICSM*, pp. 320-329, 2006.

[182] TPTP Tracing and profiling tool. Available: http://www.eclipse.org/projects/project.php?id=tptp.performance, Last Accessed on 16/02/2013.

[183] XML4Profiling. Available: http://www.eclipse.org/tptp/platform/documents/resources/profilingspec/XML4Profiling.htm, Last Accessed on 16/02/2013.

[184] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", Proceedings of *FOSE*, pp. 85-103, 2007.

[185] B. Taitelbaum*, et al.*, "Software Assurance By Synergistic Static And Dynamic Analysis", Proceedings of DSN, pp. 1-11, 2008.

[186] G. Rothermel*, et al.*, "Prioritizing test cases for regression testing", *IEEE Transactions on Software Engineering ,* vol. 27, pp. 929-948, 2001.

[187] N. Ayewah*, et al.*, "Instrumenting Static Analysis Tools on the Desktop", Microsoft Research, Tech Report, MSR-TR-2010-17, pp. 1-9, 2010.

[188] D. Poshyvanyk*, et al.*, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering,* vol. 33, pp. 420-432, 2007.

[189] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems", Proceedings of SoftVis,pp. 95-104, 2006.

[190] K. Kontogiannis*, et al.*, "Comprehension and Maintenance of Large-Scale Multi-Language Software Applications", Proceedings of *ICSM*, pp. 497-500, 2006.

[191] V. R. Basili*, et al.*, "Experimentation in software engineering", *IEEE Transactions on Software Engineering,* vol. SE-12, pp. 733-743, 1986.

[192] F. Shull*, et al.*, Eds., "*Guide to Advanced Empirical Software Engineering"*. London: Springer,p. 388 , 2008.

[193] B. A. Kitchenham*, et al.*, "Evidence-based software engineering", Proceedings *ICSE*, pp. 273-281. 2004.

[194] N. E. Fenton and M. Neil, "A critique of software defect prediction models", *IEEE Transactions on Software Engineering,* vol. 25, pp. 675-689, 1999.

[195] L. C. Briand, "Software Verification — A Scalable, Model-Driven, Empirically Grounded Approach " ,Simula Research Laboratory, A. Tveito*, et al.*, Eds., ed: Springer Berlin Heidelberg, pp. 415-442, 2010.

[196] L. Briand*, et al.*, "On the application of measurement theory in software engineering", *Empirical Software Engineering,* vol. 1, pp. 61-88, 1996.

[197] L. M. Pickard*, et al.*, "Combining empirical results in software engineering", *Information and Software Technology,* vol. 40, pp. 811-821, 1998.

[198] K. El-Emam, "A Methodology for Validating Software Product Metrics", National Research Council of Canada, NRC/ERB-1076,NRC 44142,p. 39, 2000.

[199] B. A. Kitchenham*, et al.*, "Preliminary guidelines for empirical research in software engineering", *IEEE Transactions on Software Engineering,* vol. 28, pp. 721-734, 2002.

[200] D. I. K. Sjoeberg*, et al.*, "A survey of controlled experiments in software engineering", *IEEE Transactions on Software Engineering, ,* vol. 31, pp. 733-753, 2005.

[201] R. Pastor-Satorras*, et al.*, "Dynamical and correlation properties of the internet", *Phys Rev Lett,* vol. 87, p. 258701, 2001.

[202] A.-L. Barabási, "The physics of the Web", *Physics World,* vol. 14, pp. 33-38, 2001.

[203] S. Elbaum*, et al.*, "Test case prioritization: a family of empirical studies", *IEEE Transactions on Software Engineering,* vol. 28, pp. 159-182, 2002.

[204] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap", Proceedings of ICSE-FOSE, pp.35-46, 2000.
[205] C. Zhang and H.-A. Jacobsen, "Efficiently mining crosscutting concerns through random walks", Proceedings of AOSD, pp.226-238, 2007.
[206] A. Vespignani, "Evolution of Networks - From Biological Nets to the Internet and WWW", *European Journal of Physics,* vol. 25, p. 697, 2004.
[207] A. Zaidman and S. Demeyer, "Mining ArgoUML with Dynamic Analysis to Establish a Set of Key Classes for Program Comprehension," in *International Workshop on OO Reengineering*, Universiteit Antwerpen, 2005.
[208] J. S. Shiner and M. Davison, "Quantifying the connectivity of scale-free and biological networks", *Chaos, Solitons & Fractals,* vol. 21, pp. 1-8, 2004.
[209] JHotdraw. Available: http://www.jhotdraw.org/,Last Accessed on 16/02/2013.
[210] Google.com. *Google Chrome*. Available: http://code.google.com/chromium/,Last Accessed on 16/02/2013.
[211] ArgoUML. Available: http://argouml.tigris.org/,Last Accessed on 16/02/2013.
[212] SEMERU SOFTWARE MAINTENANCE BENCHMARKS DATA. Available: http://www.cs.wm.edu/semeru/data/benchmarks/,Last Accessed on 16/02/2013.
[213] JabRef. Available: http://jabref.sourceforge.net/,Last Accessed on 16/02/2013.
[214] muCommander. Available: http://www.mucommander.com/,Last Accessed on 16/02/2013.
[215] SEMERU. Available: http://www.cs.wm.edu/semeru/index.html, Last Accessed on 16/02/2013.
[216] Tau tool suite. Available: https://www.alcf.anl.gov/resource-guides/tuning-and-analysis-utilities-tau, Last Accessed on 16/02/2013.
[217] JVMTI. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html, Last Accessed on 16/02/2013.
[218] Valgrind. *Valgrind*. Available: http://valgrind.org/, Last Accessed on 16/02/2013.
[219] Intel VTune. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe
[220] Common Language Runtime. Available: http://msdn.microsoft.com/en-us/library/cc265151%28v=vs.95%29.aspx, Last Accessed on 16/02/2013.
[221] Java Virtual Machine. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/index.html, Last Accessed on 16/02/2013.
[222] PE Format (Windows). Available: http://msdn.microsoft.com/en-gb/library/windows/desktop/ms680547%28v=vs.85%29.aspx, Last Accessed on 16/02/2013.
[223] Visual Studio Profiler. Available: http://msdn.microsoft.com/en-us/library/aa985641.aspx, Last Accessed on 16/02/2013.
[224] AQTime. Available: http://smartbear.com/products/qa-tools/application-performance-profiling, Last Accessed on 16/02/2013.
[225] P. Roy. *Slimtune*. Available: http://code.google.com/p/slimtune/,Last Accessed on 16/02/2013./
[226] MSVC Hookit. Available: http://msdn.microsoft.com/en-us/library/c63a9b7h.aspx, Last Accessed on 16/02/2013.
[227] K. B. Ferreira*, et al.*, "Characterizing application sensitivity to OS interference using kernel-level noise injection", Proceedings of Supercomputing, Article No. 19, 2008.
[228] RIGI Standard format. Available: http://www.program-transformation.org/Transform/RigiRSFSpecification, Last Accessed on 16/02/2013.

[229] V. Batagelj, *et al. Pajek*. Available: http://vlado.fmf.uni-lj.si/pub/networks/pajek/, Last Accessed on 16/02/2013.
[230] Graphviz. Available: http://www.graphviz.org/, Last Accessed on 16/02/2013.
[231] JUNG. Available: http://jung.sourceforge.net/, Last Accessed on 16/02/2013.
[232] jhotdraw. *jhotdraw*. Available: http://www.jhotdraw.org/, Last Accessed on 16/02/2013.
[233] NetBeans. *NetBeans*. Available: http://netbeans.org/, Last Accessed on 16/02/2013.
[234] Linear Transformation. Available: http://mathworld.wolfram.com/LinearTransformation.html, Last Accessed on 16/02/2013.
[235] D. Wheeler. *SLOCCount*. Available: http://www.dwheeler.com/sloccount/, Last Accessed on 16/02/2013.
[236] C. Reis and S. D. Gribble, "Isolating Web Programs in Modern Browser Architectures", Proceedings of Eurosys, pp. 219-232, 2009.
[237] GoogleChromeExperiments Arcade Fire. Available: http://www.chromeexperiments.com/arcadefire/, Last Accessed on 16/02/2013.
[238] I.E. Speed Test Fish Bowl. Available: http://ie.microsoft.com/testdrive/Performance/FishBowl/, Last Accessed on 16/02/2013.
[239] Google Chrome Multi-process Architecture. Available: http://www.chromium.org/developers/design-documents/multi-process-architecture, Last Accessed on 16/02/2013.
[240] G. A. Miller, "The Magical Number Seven, Plus or Minus Two Some Limits on Our Capacity for Processing Information", *Psychological Review* vol. 101, pp. 343-352, 1956.