

Defect Localization using Dynamic Call Tree

Mining and Matching

and

Request Replication: An Alternative to QoS-aware

Service Selection

DEFECT LOCALIZATION USING DYNAMIC CALL TREE
MINING AND MATCHING
AND
REQUEST REPLICATION: AN ALTERNATIVE TO QOS-AWARE
SERVICE SELECTION

BY
ANIS YOUSEFI, M.Sc.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

© Copyright by Anis Yousefi, November 2013

All Rights Reserved

Doctor of Philosophy (2013)
(Computing & Software)

McMaster University
Hamilton, Ontario, Canada

TITLE: Defect Localization using Dynamic Call Tree Mining and
Matching
and
Request Replication: An Alternative to QoS-aware Ser-
vice Selection

AUTHOR: Anis Yousefi
M.Sc. (Software Engineering)
Sharif University of Technology, Tehran, Iran

SUPERVISOR: Dr. Alan Wassylng and Dr. Douglas G. Down

NUMBER OF PAGES: xxii, 246

To my beloved husband and best friend, Majid, who always supports and inspires me to excel in every aspect of my life; To my dearest parents, Mitra and Naser, whose teachings and sacrifices have made me who I am today; To my brother, Rashid, the sweetest friend of all times; And, to the kindest grandparents in the world.

Abstract

This thesis is concerned with two separate subjects; (i) Defect localization using tree mining and tree matching, and (ii) Quality-of-service-aware service selection; it is divided into these parts accordingly. The underlying question addressed in the defect localization part is how can we use the dynamic call graphs representing different executions of a software system to identify root cause of a failure? This problem is broken up into two steps: (a) identifying a method as a starting location to search for the root cause, and (b) searching relevant parts of the call graph of the failing execution to find the failure's root. We focus on step (a) in this thesis. In the service selection part, we investigate how we can employ the inherent variability of service qualities to deliver better quality of service with less cost.

Defect localization using tree mining and tree matching. In the first part of this thesis we present a novel technique for defect localization which is able to localize call-graph-affecting defects (i.e., defects that make an execution diverge from the expected path, thus creating unexpected dynamic call graphs), using tree mining and tree matching techniques. In this approach, we assume a set of successful executions, and a failing execution, where some requested task is not carried out as expected. We first mine frequent patterns (i.e., subtrees) from the set of trees representing call graphs of successful executions. Then, we associate the extracted patterns

with different functionalities of the system. Next, we use a select set of patterns as a reference to compare to the call tree representation of the failing execution. Finally, we present a report, indicating method calls that are suspicious of being related to the failure. The proposed defect localization technique is implemented as a prototype and evaluated using four subject programs of various sizes, developed in Java or C. Our experiments show comparable results to similar defect localization tools, but unlike most of its counterparts, it does not require the availability of multiple failing executions to localize the defects. We believe that this is a major advantage, since it is often the case that we have only a single failing execution to work with. Potential risks of the proposed technique are also investigated.

Quality-of-service-aware service selection. In the second part of this thesis we present an alternative strategy for service selection in service oriented architecture, which is able to provide services with better quality for less cost. This approach takes advantage of the inherent variability of non-functional properties of services and mathematically shows that choosing a number of functionally-equivalent services to perform a task can provide higher quality of service, as compared with choosing a single service. Incorporating this characteristic, the proposed *Request Replication* technique suggests replicating a client's request over a number of cheap, low quality services to gain the required quality of service. The applicability of this approach is illustrated using a number of examples. Following this approach, we also present a number of suggestions about how service providers should advertise non-functional properties of their services.

Acknowledgments

This work would not have been possible had I not have the help and support of many wonderful people. I would like to sincerely thank: My supervisors, Dr. Alan Wassyng and Dr. Douglas G. Down for their constant support and guidance throughout this research. Their knowledge and insights has always enlightened my way in the most difficult times. Knowing them and working with them has been a great pleasure and a priceless experience for me. My committee members, Dr. Thomas S. E. Maibaum and Dr. Emil Sekerinski who have always supported and guided my work. My former supervisor Dr. Kamran Sartipi with whom I initially started this research project. Hosein Safyallah whose research inspired the current work. I also used some pieces of code provided by him. My wonderful collaborators at Systemware Innovation Corporation (SWI): Jim Picha, David Tremaine, Helen Xie, Natalia Lizon, Hamaad Shah, Paul Thorn, Arghavan Arjmand, and all my wonderful friends at SWI who I greatly enjoyed knowing and working with. McMaster faculty Chris George, Dr. Antoine Deza, Dr. Ryszard Janicki, Dr. Wolfram Kahl who guided me through difficulties, and patiently answered my questions. My wonderful friends at the department of Computing and Software.

Abbreviations

CDF	Cumulative Distribution Function
DFD	Done For Day
DFT	Dynamic Feature Traces
DNF	Disjunctive Normal Form
EF-ICF	Element Frequency-Inverse Concern Frequency
FIX	Financial eXchange Protocol
GCC	GNU Compiler Collection
ID	Identifier
NFP	Non-Functional Property
PDF	Probability Density Function
PID	Process Identifier
PMF	Probability Mass Function
QoS	Quality of Service
SLA	Service Level Agreement
SLO	ServiceLevel Objective
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol

SPR	Scenario-based Probabilistic Ranking
SR	Software Reconnaissance
SVM	Support Vector Machine
TF-IDF	Term Frequency Inverse Document Frequency
TID	Thread Identifier
TPTP	Eclipse Test and Performance Tools Platform
UI	User Interface
WSLA	Web Service Level Agreement
WSOL	Web Services Offerings Language
XOR	eXclusive OR

Contents

Abstract	v
Acknowledgments	vii
Abbreviations	ix
List of Tables	xvii
List of Figures	xix
1 Thesis Summary	1
1.1 Part One: Defect Localization using Tree Mining and Tree Matching	1
1.2 Part Two: Quality-of-Service-aware Service Selection	4
1.3 Thesis Outline	5
2 Introduction to Defect Localization	7
2.1 Research Objective	9
2.2 Approach in a Nutshell	11
2.3 Thesis Contributions	11

3	Defect Localization Background	15
3.1	Dynamic Analysis and Execution Tracing	15
3.2	Defect Localization	19
3.2.1	Bugs, Defects, Infections and Failures in Software	19
3.2.2	Software Testing	23
3.2.3	Software Debugging and Defect Localization	25
3.3	Feature Location	27
3.4	Data Mining	30
3.4.1	Program Call Graphs	31
3.4.2	Call Graph Reduction	32
3.4.3	Mining Frequent Itemsets	35
3.4.4	Mining Frequent Sequences	38
3.4.5	Mining Frequent Subtrees	38
3.4.6	Approximate Tree Matching	40
4	Defect Localization Literature	43
4.1	Dynamic Approaches	44
4.1.1	Delta Debugging Based	44
4.1.2	Program Slicing Based	46
4.1.3	Program Spectra Based	49
4.1.4	Discussion of Spectra-Based Approaches	57
4.1.5	Other Approaches	65
5	Defect Localization using Dynamic Call Tree Mining and Matching	67
5.1	Overview	67

5.1.1	Target Problem	68
5.1.2	Approach	71
5.1.3	Steps in Proposed Technique	74
5.2	Tracing	75
5.2.1	Instrumentation and Tracing	75
5.2.2	Features, Scenarios, and Test Cases	77
5.2.3	A Special Case: Distributed Tracing in SOA-based Systems	80
5.3	Constructing and Reducing Dynamic Call Trees	86
5.4	Mining Frequent Subtrees	90
5.4.1	Definitions	91
5.4.2	Related Work	92
5.4.3	Proposed Algorithm	93
5.5	Pattern Analysis	98
5.5.1	Pattern Semantics	98
5.5.2	Pattern Types	101
5.5.3	Identifying f -specific Candidate Patterns	102
5.5.4	Ranking Candidate Patterns	104
5.6	Defect Localization	114
5.6.1	Defect Types and their Effect on Dynamic Call Trees	114
5.6.2	Approximate Pattern Matching	116
5.6.3	Analysis of Matching Results	119
5.7	Discussion	126
5.7.1	Benefits	127
5.7.2	Risk Analysis	128

6	Evaluation	131
6.1	Prototype Implementation	132
6.2	Subject Programs	134
6.2.1	<i>FIXImulator</i>	136
6.2.2	<i>Weka</i>	138
6.2.3	<i>NanoXML</i>	139
6.2.4	<i>Print_tokens</i>	140
6.3	Evaluation Measures	141
6.4	Instrumentation	143
6.5	Localizing Defects in <i>FIXImulator</i>	145
6.5.1	Test Cases and Test Oracles	145
6.5.2	Tracing	146
6.5.3	Pattern Mining	151
6.5.4	Pattern Analysis	153
6.5.5	Defect Localization	156
6.6	Localizing Defects in <i>Weka</i>	158
6.7	Localizing Defects in <i>NanoXML</i>	161
6.8	Localizing Defects in <i>Print_tokens</i>	163
6.9	Discussion	167
6.9.1	Lessons Learned	167
6.9.2	Challenges of Experimental Evaluation	169
7	Conclusion and Future Work	171
7.1	Summary	172
7.2	Discussion	172

7.3	Future Work	175
8	Request Replication: An Alternative to QoS-aware Service Selection	179
8.1	Introduction	179
8.2	Service Selection in SOA	182
8.2.1	QoS-aware Service Selection	183
8.2.2	QoS Negotiation	184
8.3	Related Work	185
8.4	Proposed Request Replication Strategy	188
8.4.1	Motivating Example	188
8.4.2	General Approach	190
8.4.3	Request Replication	191
8.4.4	Motivating Example Revisited	196
8.4.5	Discussion	197
8.5	Revisiting Service Advertisements	201
8.6	Conclusion	203
	Bibliography	205
A	Risk Analysis	229
A.1	Tracing	229
A.1.1	Instrumenting Target System	229
A.1.2	Running Test Cases on Instrumented System to Get Execution Traces	230
A.2	Constructing and Reducing Dynamic Call Trees	231

A.2.1	Constructing Dynamic Call Trees	231
A.2.2	Reducing Dynamic Call Trees (Iteration Reduction)	231
A.3	Mining Frequent Subtrees	232
A.4	Pattern Analysis	233
A.5	Defect Localization	233
A.5.1	Pattern Matching	233
A.5.2	Analysis of Matching Results	234
A.6	General Risks	235
B	Pattern Analysis using Association Rules Mining	237
B.1	Main Idea	237
B.2	Generalized Association Rules Mining	239
B.3	Discussion	240
C	Dealing with Concurrency	243
C.1	Concurrency Defects	243
C.2	Select Related Work in Localizing Concurrency Defects	244
C.3	The Proposed Idea	245

List of Tables

4.1	Suspiciousness formulations presented in literature	50
4.2	Related spectra-based approaches	58
6.1	Code counts for FIXImulator	138
6.2	Code counts for Weka	139
6.3	Code counts for NanoXML	140
6.4	Code counts for print_tokens	141
6.5	FIX Protocol Messages	153
6.6	Mapping test cases to features they exercise	155
6.7	Results of defect localization for FIXImulator	158
6.8	Results of defect localization for Weka	159
6.9	Eichinger et al.'s results for Weka (Eichinger <i>et al.</i> , 2010b)	160
6.10	Results of defect localization for NanoXML	161
6.11	Results of defect localization for print_tokens	164
B.1	Example of a test-pattern-feature table	238

List of Figures

3.1	Overview of principal elements in dynamic analysis (Cornelissen, 2009)	17
3.2	(a) Original call graph (b) call graph with a structure-affecting defect (c) call graph with a frequency-affecting defect (Eichinger, 2011) . . .	22
3.3	(a) An unreduced call graph (b) simple total reduction (c) total reduction with edge weights (d) total reduction with temporal edges (e) Unordered 0-1-m reduction (f) ordered 0-1-m reduction (g) subtree reduction (h) an unreduced call graph (i) direct reduction (j) an unreduced call graph (k) indirect reduction (Eichinger, 2011)	34
3.4	(a) An unreduced call graph (b) omitting isomorphic substructures in direct sequence (c) omitting isomorphic chain of substructures in direct sequence (d) repeating (b) and (c) for parent nodes until no iterations are left	36
3.5	(a) Sample tree, (b) a bottom-up subtree, (c) an induced subtree, and (d) an embedded subtree	40
3.6	A pattern and approximate matches in a call tree: (a) pattern (b) insertion (c) deletion (d) substitution	42
4.1	An imbalanced class distribution where 99 out of 100 tests pass and only one fails	61

5.1	Sample code with structure-affecting defects and patterns associated with execution of defect-free and defective code	70
5.2	Sample dynamic call tree associated with sequential execution of tasks “x”, “y”, and “z”	73
5.3	Proposed defect localization process	74
5.4	Sample execution trace	76
5.5	Sample test case for a financial system	80
5.6	Sample SOA-based system	82
5.7	(a) Service trace files partitioned into blocks (b) block execution tree associated with (a)	84
5.8	(a) Sample trace and (b) corresponding multi-threaded call tree . . .	87
5.9	(a) Sample tree, unique IDs assigned to its subtrees, and symbolic encoding of its root method (b) the tree after calling <code>findAndRemoveRepetitions</code> once (c) the tree after all iterations are removed	89
5.10	Sample tree-set and four iterations of proposed mining algorithm . . .	97
5.11	Patterns and their relation to target feature, “Acknowledge”	99
5.12	Control structures and their corresponding patterns	100
5.13	Relation of pattern A to pattern B with regard to their methods: (a) patterns have no common methods (b) pattern A and B have some shared methods (c) pattern A includes all methods in pattern B . . .	111
5.14	(a) A pattern tree and (b) a subject tree	120
5.15	(a) pattern tree (b) subtree with matching root from subject tree . .	120
5.16	Defect localization report	125
6.1	Architecture of proposed defect localization tool	133

6.2	Architecture of target FIX-based trading program	136
6.3	Sample probe created with Probekit editor	144
6.4	Filters associated with sample probe	144
6.5	Sample test case	146
6.6	<i>Banzai</i> sending a new buy order to <i>FIXImulator</i>	147
6.7	<i>FIXImulator</i> receiving a new buy order from <i>Banzai</i> and sending a number of messages	148
6.8	Running a test script using JUnit	149
6.9	Excerpt of a sample trace file	150
6.10	XML file holding discovered patterns	151
6.11	Relation between total pattern count and number of test cases	152
6.12	Ranked list of candidate feature-specific patterns for feature <i>F2.5</i>	156
6.13	Feature-specific pattern for feature <i>F2.5</i> , display of message done for day	157
6.14	Results of comparing <i>F2.5</i> -specific patterns with call tree of failing execution	157
6.15	Comparison of a number of approaches using Siemens suite subject provided by Yu et al., (Yu <i>et al.</i> , 2011), with additional data points from our tool	165

Chapter 1

Thesis Summary

This thesis is divided into two parts. The first part concerns *defect localization using tree mining and tree matching*, and the second part *quality-of-service-aware service selection*. In the first part we address this research question: how can we incorporate run time information about various executions of a software system (specifically dynamic call graphs) to identify the root cause of a failure? In the second part we investigate the following: how can we employ the inherent variability of service qualities to deliver better quality of service with less cost?

1.1 Part One: Defect Localization using Tree Mining and Tree Matching

In the first part of this thesis (Chapters 2 to 7) we address the problem of software defect localization. Defect localization is one of the major challenges that software

providers face. Software systems frequently demonstrate unexpected behavior or provide incorrect results during their executions, even after they are released. Defect localization refers to part of the debugging process which is responsible for finding code segments causing such unexpected behaviors or results. This task in general requires vast understanding of the software system's usage domain, its code, and specifically the link between source code constructs and the system's run time behavior. Understanding and recalling the code-behavior relation in today's huge systems with their distributed and concurrent algorithms is a complex, near impossible, task, unless it is supported by effective diagnostic tools. Defect localization helps software engineers downsize the problem by identifying a small potentially defective part of the code to avoid unnecessary code review when searching for the cause of a failure.

In the first part of this thesis, we present a novel technique and tool for defect localization. This technique spots methods that are possibly relevant to a failure by noting the differences between dynamic call graph of the failing execution and a select set of correct executions. Following this approach, one will be able to find and rank a set of subgraphs of the failing call graph that are likely to include the defective code. The technique also identifies suspicious methods on each candidate subgraph. Such methods can be good starting points to search for the defective code. This approach is intended for defects that cause structural changes to the dynamic call graph of an execution (i.e., call-graph-affecting defects). In other words, the dynamic call graph observed in the presence of a call-graph-affecting defect for a certain failing scenario differs from the correct (i.e., expected) call graph.

The proposed defect localization tool first applies feature location to identify the link between different functionalities of the system (also called *features*) and the code

implementing those functionalities. In this step, dynamic call trees (i.e., tree representations of call graphs) are mined to discover subtrees that are frequently executed. The discovered patterns are then linked to system functionalities using feature location techniques. Assuming that the failing functionality is known, our technique then compares patterns that are assumed to represent correct executions of the failing functionality against relevant subtrees from the call tree of the failing execution. A missing or extra method may point to a problem (e.g., a run-time exception, an incorrect branch condition, an improper change during previous debugging, a change from a previous version of the code, etc). Such a method can be used as a starting point for further root cause analysis.

The work presented in the first part of this thesis provides a foundation for further research in defect localization and feature location. A prototype of the proposed technique has been implemented and evaluated using four subject systems. The strengths and weaknesses of the tool are analyzed and a discussion of benefits, risks and possible future research tracks is provided. We claim that the proposed technique is more powerful compared with similar defect localization tools in the literature, in the sense that we require less information for our analysis. It also is more effective in dealing with systems having multiple functionalities, because we incorporate feature-location to downsize the search space. Finally, it is more informative as it indicates not only the method where the problem manifests but also the methods triggering the problematic method by identifying and ranking subtrees in the call tree of the failing execution that potentially include the problematic method.

1.2 Part Two: Quality-of-Service-aware Service Selection

The second part of this thesis (Chapter 8) presents our approach to the problem of QoS-aware service selection in service oriented architecture (SOA). Service selection, in general, refers to the process of finding services that match a client's functional requirements. QoS-aware service selection also takes into consideration the non-functional requirements when searching for proper services. The non-functional properties of a service are usually published along with the service interface in the form of QoS advertisements. Current QoS advertisements typically provide a single value to represent the distribution of a non-functional property such as response time. However, the SOA literature implies that non-functional properties such as response time have inherently high variance in their values (Gorbenko *et al.*, 2009; Zhu *et al.*, 2006), and thus representing non-functional properties with a single value does not reveal much about their actual distribution. This makes it hard for service clients to choose any selection strategy other than the conventional QoS-aware service selection which is geared for clients choosing "a single service" to serve their needs.

In the second part of this thesis, we propose a new strategy for QoS-aware service selection which takes advantage of the existing variability in QoS data to provide higher quality services with less cost compared with conventional QoS-aware service selection methods. In this method, we replicate each request over multiple independent services to achieve the required QoS. We also present a number of recommendations regarding the QoS advertisements in SOA so as to reveal more information

about underlying distributions and thus enabling more sophisticated selection strategies. We will show using various examples how this approach works and enhances conventional QoS-aware service selection.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 provides an introduction to the first part of this thesis, the problem of defect localization. Chapter 3 presents basic information that is required to understand the problem of defect localization as well as the proposed solution. We define terms and concepts in dynamic analysis, defect localization, feature location and data mining. Chapter 4 presents related literature in the domain of defect localization. Chapter 5 presents our defect localization approach. Chapter 6 provides an experimental evaluation of the proposed defect localization tool. Chapter 7 concludes the defect localization part of this thesis. Chapter 8 presents our approach and results for quality-of-service-aware service selection. This includes an introduction to the problem, background information, related work, the proposed approach and conclusions and future work. In Appendix A we discuss the risks associated with the proposed defect localization technique. Appendix B explains an alternative approach for pattern analysis which uses association rules mining. Appendix C discusses how one can potentially extend our approach to deal with concurrency defects.

Chapter 2

Introduction to Defect Localization

“Poor software quality and security remain major problems for many businesses as they grapple with a steady flow of applications, upgrades, and fixes...Even experienced programmers inject about one defect into every 10 lines of code, according to the Software Engineering Institute. If 99% of those are caught, that’s still 1000 bugs in a one million-line application” (Hulme, 2002). Software programs are built by humans and humans make mistakes. Despite available best-practices for software development such as the Rational Unified Process (RUP), specification-driven development and formal verification methods to prove the correctness of intended algorithms underlying a system, software systems remain to exhibit buggy behavior after they are released. In fact as Garrison Hoffman, a software engineer for technology consulting firm Intrasphere Technologies Inc. in New York, noted *“There’s always going to be a bug you haven’t found”* (Hulme, 2002). Too often these bugs are found by customers (Hulme, 2002).

Software verification intends to find if software fully satisfies all the expected requirements (Cheng, 2010). Static verification techniques examine software code to

discover problems primarily by checking if software meets specified requirements. In this context, techniques such as formal verification aim to check whether the algorithms underlying the system satisfy certain properties specified in a formal language. This is done using formal mathematical techniques. Static verification techniques can be of great help in examining software code, specifically code paths that are rarely executed in practice. However, static verification techniques cannot identify if a function actually does what it is supposed to do with regard to the requirements (i.e., functional correctness) (Rielly, 2011). Dynamic verification or testing is used to verify functional correctness. The idea behind dynamic testing is “to create test cases that associate specific input values with output values (expected values) and execute the tests to ensure the program is accomplishing its goals” (Rielly, 2011).

Software testing reveals the existence of defects in the code but provides no clue as to why a failure happens (i.e., the location of defects). The task of finding and fixing software bugs is usually done through in-depth code review along with testing and classical debugging (Eichinger *et al.*, 2008). This, in one hand requires extensive knowledge about the software domain of use, the underlying algorithms, and technologies and programming languages used to implement the algorithms. In the other hand, manual search is time-consuming and expensive. Program understanding tools help in building the required knowledge. Debugging tools provide a basis for faster and easier code investigation. Above all, defect localization tools lead the programmers to the location of defects with minimal pre-existing knowledge, thereby reduce the need to acquire knowledge about the system and speed up the search process.

In this work we are examining the problem of defect localization. One way to

localize defects in software is to analyze dynamic call graphs with graph-mining techniques (Di Fatta *et al.*, 2006; Liu *et al.*, 2005; Cheng *et al.*, 2009; Eichinger *et al.*, 2008, 2010a,b, 2011). Graph mining is a general technique for the analysis of graph structures (Aggarwal and Wang, 2010; Cook and Holder, 2006). In this work we intend to show how we can leverage graph mining and matching techniques to learn more about a failure and its origin.

2.1 Research Objective

Our aim in this work is to provide a tool to assist in the process of defect localization. Defect localization in general refers to identifying defective statements in the code. There are numerous static and dynamic techniques that help spot defective code. For example, approaches such as evaluating software complexity measures (Ujhazi *et al.*, 2010) and mining revision histories (Livshits and Zimmermann, 2005) can help reveal error-prone components of a system. Also, looking for suspicious code patterns (Palix *et al.*, 2010) or invariants (Abreu *et al.*, 2008) helps spotting programming defects.

In practice, programmers usually rely on testing to find software deficiencies. They run test cases on a system and resolve issues found during the execution. In this scenario, they are searching for the cause of certain failure cases instead of looking for all suspicious code in general. This approach is more to-the-point and consequently requires less effort to make the program work. In other words, instead of exhaustively searching for code deficiencies, the programmers typically seek reasons behind a certain failure. Following this practice, in this work we assume the availability of a certain failure case to be analyzed. We then search for methods in the source code of the target system which contribute to the failure.

To achieve this goal, we assume the availability of numerous successful cases (i.e., tests where the requested functionalities are carried out successfully), which we use to build knowledge about the target system’s functionalities via execution pattern mining. This is a practical assumption. A system in the testing phase often performs its functionality partially. Therefore, we can think of more successful scenarios than failure cases. Note that, a failure is not only a situation where programming errors happen but also any case where wrong results are produced or unexpected reactions to a scenario are observed, as in the case of functional testing.

As we will see in Chapter 4, a large body of research in defect localization assumes the existence of multiple failing executions. There are some that, similar to this work, deal with a single failure case (Dallmeier *et al.*, 2005; Renieris and Reiss, 2003). However, they incorporate other techniques to find the root of a failure. Most of the defect localization techniques are not designed with the complexities of today’s big systems in mind. Therefore, such approaches are not suitable for large applications. For example, approaches such as program slicing (Agrawal, 1992) and delta debugging (Zeller, 1999) do not scale well. Call graph-based techniques such as (Eichinger, 2011) are more effective when dealing with big applications. However, as we will see in Chapter 4, they: 1) usually are based on stronger underlying assumptions (i.e., assume multiple failing and multiple passing executions); 2) do not specifically plan for large multi-feature systems.

Based on the above discussion, we identified the major goal of this study to be devising a novel technique and a tool for defect localization, which:

- Finds a starting point for debugging and root cause analysis;
- Handles systems with multiple functionalities;

- Deals with defects that cause a structural change in the call graph;
- Is constrained to work with just a single failing test case;
- Assumes multiple successful test cases.

2.2 Approach in a Nutshell

The basic idea behind our defect localization technique is to combine the power of “feature location”, “frequent-pattern mining” and “pattern matching” to localize defects. We use feature location to highlight, in the dynamic call tree of an execution, which parts are related to which features. In this approach, we use frequent-pattern mining to mine patterns among dynamic call trees of successful executions. Then we analyze those patterns to discover which patterns are related to which features. A feature-specific pattern identifies method calls that are frequently observed in successful executions of a feature. It represents a fault-free execution of the feature. When the execution of a feature fails, we use the feature-specific patterns associated with that feature as a base to investigate the call tree of the failing execution. In this method, we apply approximate pattern matching to identify deviations from the base. Then, we present a ranked list of suspicious method calls, which provides a starting point to search for the root cause of a failure.

2.3 Thesis Contributions

The major contribution of this thesis is *devising a novel technique and tool for defect localization, which is able to identify structure-affecting defects (defined in Chapter 3)*

in systems with multiple functionalities, having only a single failing execution. These assumptions call for a more powerful technique than related approaches which assume the availability of multiple failing executions or deal with smaller applications. As part of our study, we also provide the following contributions:

- We discuss the challenges of distributed execution tracing in service-based systems and present an approach for aggregating execution traces.
- We devise an algorithm for mining frequent bottom-up sub trees which enhances its existing counterparts.
- We enhance the conventional notion of feature location by identifying not only the methods associated with a certain software functionality (i.e., feature), but also the sub-call-trees that have exclusively been executed as part of the execution of the feature. This improves one's understanding of how (in addition to where) a feature is implemented. In this area, we suggest new method-feature relevance formulations and a mechanism to quantify the relevance of call trees to features, based on the methods they include.
- We enhance the state of defect localization by incorporating feature location. We use feature location to highlight parts of a failing execution that are related to the failing feature, thus reducing the effort needed to look for defect roots, in systems with multiple functionalities.
- By the incorporation of pattern matching and feature location techniques, we devise a defect localization technique that is more powerful than similar call-graph-based approaches, because we eliminate the need for multiple failing cases when analyzing a failure.

- Our technique provides a more informative defect localization report. The report includes a list of suspicious method calls accompanied by their associated subtrees as well as expected alternatives. This provides insights as to where and what the defects may be, which is beneficial for root cause analysis and also when fixing defects.
- We present an experimental study of our tool in Chapter 6. We use our tool to localize pre-known defects of a number of subject systems with different characteristics. The experimental results are published in this thesis and can be used by other researchers in this domain.
- We provide a risk analysis of the proposed technique in Chapter 5 and Appendix A, which can be used to learn about the current work as well as to identify future directions of research.

Chapter 3

Defect Localization Background

Our goal in this chapter is to explain the basic concepts that are required for understanding the remainder of this thesis. The proposed approach in this thesis relates to the areas of “dynamic analysis”, “defect localization”, “feature location” and “data mining”. The following sections introduce these areas.

3.1 Dynamic Analysis and Execution Tracing

Dynamic analysis is one of the most popular techniques incorporated into defect localization and feature location tools. Specifically, the proposed defect localization technique in this thesis is based on dynamic analysis and execution tracing. Therefore, in this section we examine the basics of these techniques in more details.

Most maintenance and debugging activities (e.g., adding a new functionality to a system, migrating to a new environment, troubleshooting, etc) require deep understanding of the source code and algorithms implementing a software system. *Program comprehension*, or understanding a software’s source code, is one of the major

challenges in software engineering, especially when adequate documentation is not available. The complexity of this task has inspired extensive research in this area which consequently has led to the development of numerous supporting techniques such as execution trace analysis, architecture reconstruction, and feature location. These techniques take advantage of *static* and *dynamic* analyses to examine a target system. Static analysis is the study of source code of a system without executing it while dynamic analysis refers to the study of software execution. The advantage of static analysis is its completeness as it deals with the source code which represents the full description of the system. One drawback of static analysis is that it does not capture dynamic aspects of the system. For example, how variable assignments change the run time behavior of the system. In contrast, dynamic analysis is able to deal with run time aspects of the system, however it is not complete.

Figure 3.1, adapted from (Cornelissen, 2009), illustrates the principal elements of dynamic analysis. The process typically involves running a set of *task scenarios* (or *scenarios* for short) on a software system (called the *target system*) which is *instrumented* to print out information about executions. The following introduces these concepts in more details.

In dynamic analysis we study the execution of a target system. This typically starts by running a set of task scenarios on the target system. A task scenario is a sequence of user-system interactions to perform a certain task. For example, in a banking system to withdraw money from one's account:

1. the client initiates a log-in request, providing their credentials;
2. the banking system authenticates the client;
3. the client requests a withdrawal for a certain amount from their account;

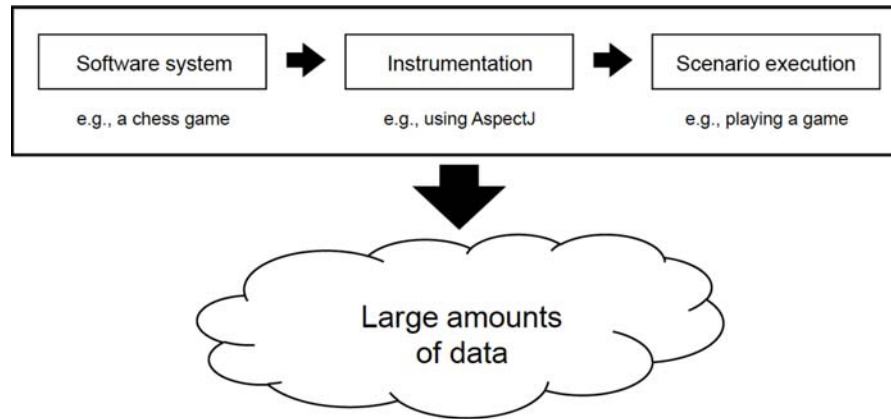


Figure 3.1: Overview of principal elements in dynamic analysis (Cornelissen, 2009)

4. the banking system checks the balance of the client's account and provides the requested cash if approved.

As you see, in this scenario there are steps to be accomplished by the client and the banking system in order to fulfill a certain task.

To study the execution of a target system one typically incorporates *execution tracing*. An *execution trace* (or *trace* for short) is a record of dynamic characteristics of a system in an execution, including values of variables, messages communicated between different components of the system, the code executed by the system, timing statistics, and other information necessary to debug and tune the system (Cole, 2009). The process of recording execution traces is referred to as execution tracing, or *tracing* for short. Different mechanisms such as *interpretation* (e.g., using the Virtual Machine in Java) or *instrumentation* (e.g., using AspectJ (Kiczales *et al.*, 2001)) have been proposed for capturing execution traces.

Software instrumentation usually refers to inserting print-out commands at certain points (e.g., every method entry, every method exit, etc.) in the binary or source code of a program to record certain information (e.g., line of code executed, name of method called, time of an event, etc.) about the system being executed. Instrumentation tools also provide the ability to filter out certain packages, classes, or methods in the resulting traces. There are many ways to instrument source code or binary of a software system. One can incorporate instrumentation tools such as Aprobe (Cole, 2009) or TPTP (Mehregani, 2006) or provide custom code to instrument a target system (e.g., AspectJ). Compilers such as GNU GCC (Stallman and the GCC Developer Community, 2003) also provide tracing capabilities.

Experienced developers insert instrumentation code in their application while the application is being developed. However, new issues always arise after the development process is completed, which require different execution information in order to be resolved. Therefore, there is always a need for updating the instrumentation code to match the current requirements. This led developers to use *probing* mechanisms. A *probe* is an instrumentation code which is not part of the application's source code. It is kept along with the source code in specific libraries and is inserted into the source code upon need. Probing frameworks such as TPTP Probekit (Mehregani, 2006) and Aprobe (Cole, 2009) let developers create probes and add them to the program's code *statically*, similar to the conventional software instrumentation, or *dynamically* by inserting calls to probes into the application's binary code at run time.

In our experiments in this thesis we used TPTP Probekit (Mehregani, 2006) to instrument Java-based systems and GNU GCC compiler's probing capabilities to instrument C code. Details of this are provided in Section 6.4.

3.2 Defect Localization

As part of the software development process, software engineers need to ensure that a program's implementation meets the requirements that has guided its design and development. Finding and fixing defects in the source code of a program are among the responsibilities of software *testing* and *debugging*. *Defect localization* is a part of the debugging process that locates defects in the source code of a software. This is required before any attempts to fix the defects can be made. The following sections discuss related concepts in software testing and defect localization.

3.2.1 Bugs, Defects, Infections and Failures in Software

In the literature of defect localization, different terminology is used to refer to fairly similar concepts. For example, one may use any of the terms *defect* or *fault* to cite an incorrect block of program code. Also, *defect localization* and *fault localization* are used interchangeably to refer to the process of locating such incorrect code. In addition, terms such as *fault*, *bug* and *error* are used in different contexts to bring up rather disparate concepts. For example, a *bug* can refer to an incorrect program code (“This line is buggy”), state (“This pointer, being null, is a bug”), or execution (“The program crashes; this is a bug”). Zeller (Zeller, 2009) and others, suggest avoiding the use of these terms and replacing them with more precise terminology. To avoid any ambiguity in this thesis, we will use Zeller's terminology, as follows:

- **Defect** is an incorrect section of program code.
- **Infection** is an incorrect program state which is usually triggered by defects.

- **Failure** is an observable incorrect program behavior, which either causes the program to crash or provides unexpected results.

A defect can cause an infection which can itself result in a failure. When one observes a failure, one usually tracks the infection to find and fix the defect. A *correct, successful, or passing* execution is an execution which leads to a correct program behavior and an *incorrect, unsuccessful, or failing* execution is an execution which results in an observable diversion from the expected behavior.

The following definitions are borrowed from Eichinger (Eichinger, 2011) and explain different types of failing behavior:

- **Crashing and non-crashing defects:** Crashing defects lead to an unexpected termination of the program. Examples include null pointer exceptions and division by zeros. Non-crashing defects, lead to wrong results. Crashing defects typically present a stack trace. The stack trace may or may not provide useful hints about where the infection has occurred. For non-crashing defects, there are usually no hints about what went wrong during the execution (Liu *et al.*, 2005; Lo *et al.*, 2009). The proposed defect localization technique in this thesis is able to handle both crashing and non-crashing defects.
- **Occasional and non-occasional failures:** Occasional failures are those which occur with some but not with all input data. Localizing defects associated with occasional failures is particularly difficult for the following reasons: 1) they are harder to reproduce, thus fewer failing executions can be attributed to them; 2) it is more probable that they still exist after early testing of the software, where non-occasional failures are usually detected. In this work we are focusing

on occasional failures that happen so rare that not many failing cases can be attributed to them. However, there are no obstacles in using our approach to localize non-occasional failures in the same way as for occasional failures.

- **Call-graph-affecting defects (including structure- and call-frequency-affecting defects):** Assume that exercising a specific scenario on a defect-free program results in the execution of dynamic call graph g . Then, assume that we embed a defect into the program, run the same scenario, but get a different dynamic call graph g' . Such a defect is called a call-graph-affecting defect. In this case, call graph g is called the *expected, original* or *correct* call graph and g' the *defective* call graph. There are two types of call-graph-affecting defects. A structure-affecting defect is one which result in some missing or additional parts in the call graph of the failing execution as compared with the expected call graph. A call-frequency-affecting defect (frequency-affecting defect for short) is one which affects the number of invocations of a method. Figure 3.2, borrowed from Eichinger (Eichinger, 2011), illustrates both kinds of defects. In this figure, (a) illustrates a dynamic call graph which represents a correct execution of the program given in Listing 3.1, also borrowed from Eichinger (Eichinger, 2011). Assuming (a) as the original graph, graph (b) represents the execution of the same program in the presence of a structure-affecting defect such as a defective `if`-condition in the `main` method. Compared with (a), in (b) the call of method `A` from method `main` is missing. Graph (c) represents the execution of the same program in the presence of a frequency-affecting defect such as a defective loop condition or a defective `if`-condition inside the loop in method `B`. This leads to the increased number of calls of method `A` in (c) as compared with (a).

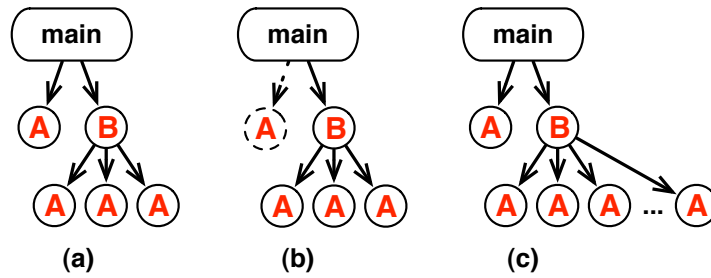


Figure 3.2: (a) Original call graph (b) call graph with a structure-affecting defect (c) call graph with a frequency-affecting defect (Eichinger, 2011)

Listing 3.1: Sample Java program

```

1 public class Example {
2     public static void main(String [] args) {
3         int x = 0;
4         if (x > 0)
5             A();
6         B();
7     }
8     public void A() {
9         System.out.println("inside method A");
10    }
11    public void B() {
12        for (int i = 0; i < 3; i++)
13            A();
14    }
15 }

```

- **Call-graph and dataflow-affecting defects:** In contrast to call-graph-affecting defects, dataflow-affecting defects manifest themselves by changing the data exchanged between program components. In this thesis, we focus on call-graph affecting (specifically structure-affecting) defects. However, dataflow-affecting defects may also affect the call graph as a side effect and thus be localizable using our approach.

3.2.2 Software Testing

The aim of *testing* is to ensure that programs provide the intended functionality. Part of this process is *software verification*, in which the expected behavior of a target program is specified as a set of *software tests* (also called *test cases*), and verified against the program. A test case consists of program inputs and expected outputs (Beizer, 1990), which help determining whether the program under test is working as expected. Program inputs include system configurations, program parameters and files and user input read by the program, and the expected output includes everything that is produced by the program, such as files written and output displayed on screen (Eichinger, 2011). For example, a sample test case for a banking system may check for the system’s reaction when the client’s balance is less than the required withdrawal amount. A test case usually has a *test scenario* which, similar to a task scenario used in dynamic analysis, identifies steps to be followed during the test, expected visible interactions between parties, and expected reaction of the system. A collection of test cases that are intended to be used to test a software system in order to show that it has some specified set of behaviors is called a *test suite*.

There is a complex field of research related to designing software tests. It typically aims at covering many different non-overlapping executions of a program which execute possibly large parts of the source code (Eichinger, 2011). This is called *test coverage analysis* and studies the coverage of tests over code, requirements, features, etc. In this work we are interested in code coverage analysis. *Code coverage* is a measure describing the degree to which the source code of a program has been tested. There are a large number of code coverage criteria, the main ones being (Cornett, 2010):

- *Statement Coverage.* This metric reports whether or not each executable statement is reached during testing.
- *Decision Coverage.* This metric reports whether Boolean expressions in control structures such as the `if`-statements and `while`-statements were evaluated to both `true` and `false` in the test suite. In decision coverage, the entire Boolean expression is considered one `true-or-false` predicate regardless of whether it contains logical-`and` or logical-`or` operators. Moreover, this metric is used to monitor the coverage of `switch-case`-statements, exception handlers, and all points of entry and exit.
- *Condition Coverage.* This metric reports whether each condition in Boolean expressions is evaluated to both `true` and `false`. A condition is an operand of a logical operator which does not contain any logical operators.
- *Path Coverage.* This metric reports whether each of the possible paths in each function in the code have been executed in tests. A path is a unique sequence of statements and branches in a function from its entry to its exit.

To identify the expected output and compare it with the actual output of the program under test, one usually relies on *test oracles* (Howden, 1978). A test oracle typically identifies the expected output for a given input. Some also refer to a program which produces the correct output and compares it with the actual output as the test oracle (Eichinger, 2011). Test oracles are used in the testing process to decide whether a certain execution leads to any failures. A test that runs as expected and successfully completes its execution is called a *successful* or *passing* test and one that encounters problems during its execution is called *unsuccessful* or *failing*. For simplicity we refer to *passing/failing test cases* as *passing/failing cases* in this thesis. Later in this thesis we use terms *passing traces* and *failing traces* to refer to traces associated with passing and failing test cases, respectively.

In this thesis we assume that both test cases and test oracles are available, as we focus on the defect-localization step. This assumption is reasonable, since testing is an inherent part of modern software development (Sommerville, 2010), and developers invariably have a test suite for their applications. The quality of the test suites may vary, but all we claim at this stage is simply the existence of a test suite.

3.2.3 Software Debugging and Defect Localization

The aim of *software debugging* is to find and fix the defects that cause failures. Debugging has also been described as the process of relating a failure first to an infection and then to a defect (Zeller, 2009). Software debugging includes everything from dealing with test cases and observing program executions, to localizing defects and ultimately fixing them (Eichinger, 2011). In this thesis, we develop a novel technique for the defect localization part of debugging. That is, we aim at helping

software developers in localizing defects once a failure has occurred.

Defect localization techniques are either *static* or *dynamic* (Binkley, 2007). Dynamic techniques rely on the analysis of program executions while static techniques do not require any execution. An example for a static technique is source code analysis, where the software's source code or graph representations of it are examined to identify potential location of defects. Program components with suspicious values of specific measures or defect-prone programming patterns are good hints for statically localizing defects. However, static approaches typically lead to many false positive warnings, and they have difficulties discovering run-time failures (Rutar *et al.*, 2004).

Dynamic techniques usually trace some information during the execution of a software program and incorporate those traces in the process of defect localization. They analyze program executions and typically compare specific characteristics such as method call counts in correct and failing executions (Eichinger, 2011). Dynamic traces may include values of variables, messages exchanged between different parties, code segments executed during the run, etc. Dynamic defect localization approaches use different information derived from executions, as well as different methodologies to perform defect localization. Also the smallest unit that can be reported as defective is different. Chapter 4 provides a detailed discussion of dynamic defect localization literature.

Although, the terms *defect localization* and *root cause analysis* are used interchangeably in the literature, in this work we use them in different contexts. By defect localization we mean finding code relevant to the problem i.e., on the path to the root cause of the failure. However, by root cause analysis we mean determining the most basic source of a problem, i.e., the root cause. The latter usually involves

considering the correlation between code components, to track the symptoms of a failure back to the root cause.

3.3 Feature Location

There are different definitions provided for the term “feature” in the literature. In software systems, a *feature* represents “a functionality that is defined by requirements and accessible to developers and users” (Dit *et al.*, 2011). In this work we use the definition provided by Dudar (Dudar, 2012), that is “a product’s discrete unit of unique and attractive functionality that delivers measurable benefit to customers” . The latter definition implies that a feature must be a component of the work required to achieve a product’s goals.

Wilde and Scully (Wilde and Scully, 1995) pioneered the field of *feature location* by introducing their Software Reconnaissance tool in 1995. Feature location is defined as finding a primary location in the source code that implements a specific functionality of a software system (Biggerstaff *et al.*, 1994; Rajlich and Wilde, 2002). It involves identifying the relations between system functionalities and different parts of the source code, and has proven to be a popular research interest to the present day (Cornelissen, 2009).

As explained by Dit *et al.*, (Dit *et al.*, 2011), feature location is one of the most important activities performed by software engineers for software maintenance, evolution and debugging. This includes tasks such as adding new features to programs, improving existing functionalities, and removing bugs from existing features. Dit *et al.*, also claim that no maintenance activity can be accomplished without first locating

the code that is relevant to the task at hand, i.e., feature location. Software engineers incorporate feature location to identify where in the code the first changes to complete a task need to be made. Then starting from the code segments identified by feature location, they incorporate impact analysis to find all blocks of code affected by the change. The following example provided by Dit et al., shows how one can take advantage of feature location for the purpose of debugging. “Alice is a new developer on a software project, and her manager has given her the task of fixing a bug that has been recently reported. Since Alice is new to this project, she is unfamiliar with the large code base of the software system and does not know where to begin. Lacking sufficient documentation on the system and the ability to ask the codes original authors for help, the only option Alice sees is to manually search for the code relevant to her task. However, a manual search of a large amount of source cods, even with the help of tools such as textual-pattern-matchers or an integrated development environment, can be frustrating and time-consuming. Recognizing this problem, software engineering researchers have developed a number of feature location techniques to aid programmers in Alice’s position”.

Different types of software analysis have been employed to identify blocks of code associated with software features. The proposed techniques are all unique in terms of their input requirements, how they locate a feature’s implementation, and how they present their results. The following, borrowed from Dit et al., (Dit *et al.*, 2011), explains the most common types of analysis used for feature location: *dynamic*, *static*, and *textual*.

Dynamic analysis refers to examining the execution of a software system. In

dynamic analysis based feature location a feature is observed during run time. Typically, one or more feature-specific scenarios (i.e., those that exercise the desired feature) are developed. Then, the scenarios are executed on the system which has previously been instrumented, and execution traces are collected. These traces record information about the code that has been executed. Once the traces are collected, feature location can be performed in different ways. The traces can be compared with other traces where the feature has not been invoked to find the pieces of code that have only been observed in feature-specific traces (Eisenbarth *et al.*, 2003; Wilde and Scully, 1995). Alternatively, one can analyze the frequency of the observation of different pieces of code in feature-specific traces and those that do not invoke the desired feature to locate the feature's implementation (Antoniol and Gueheneuc, 2006; Eisenberg and De Volder, 2005; Safyallah and Sartipi, 2006).

Dynamic analysis is a popular approach for feature location. This is because most features can be mapped to execution scenarios. However, dynamic analysis based approaches have some limitations: 1) Tracing can impose considerable overhead on the execution of a system. 2) The scenarios used to collect traces may not invoke all of the code that is relevant to the feature, thus some parts of the feature's implementation may not be located. 3) It may be difficult to build a scenario that invokes only the desired feature. This may cause irrelevant code to be executed (Dit *et al.*, 2011).

Static analysis refers to examining structural information such as static call relations and control or data flow dependencies. In manual feature location, developers may first identify a section of code as relevant to the desired feature, then follow program dependencies in order to find additional related code. This idea is used in some feature location approaches, such as (Chen and Rajlich, 2000). Other static

techniques such as (Robillard, 2008) may analyze the topology of the structural information to point programmers to potentially relevant code. Static analysis often overestimates what is pertinent to a feature and is prone to returning many false positive results (Dit *et al.*, 2011).

Textual analysis refers to examining the words used in source code of a system. Textual feature location approaches such as (Deerwester *et al.*, 1990; Blei *et al.*, 2003; Salton and McGill, 1986) are based on this assumption that identifiers and comments convey domain knowledge, and a feature may be implemented using a meaningful set of words throughout a software system, which makes it possible to find the feature's relevant code through proper querying. Despite the type of textual analysis used for feature location, the quality of the results is heavily dependent on the quality of the source code naming conventions and the query issued by the user (Dit *et al.*, 2011).

3.4 Data Mining

The proposed defect localization technique in this thesis uses solutions from the domain of data mining including frequent-subtree mining and approximate tree matching algorithms. In this section we provide necessary data-mining-related background information. We start with an introduction of program call graphs, then briefly overview call graph reduction techniques. Most frequent-subtree mining algorithms borrow ideas from frequent-itemset mining algorithms, so we continue this section with a review of frequent-itemset mining concepts that have also been employed in frequent-subtree mining. We conclude this section with an overview of the concepts in approximate tree matching.

3.4.1 Program Call Graphs

Graphs have been used for different purposes in software engineering. The following is borrowed from (Eichinger, 2011) and introduces graphs that are relevant to this discussion.

Control-flow graphs (Allen, 1970) are static representations of source code, illustrating all the paths that may be executed during a software execution. The vertices of a control-flow graph of a software represent its basic blocks, i.e., a group of statements that are always executed conjunctively. An edge represents control-flow changes from one basic block to another, for example due to an `if` or `for` statement. Control-flow graphs are frequently used in compiler technology.

Program-dependence graphs (Ottenstein and Ottenstein, 1984) are static graphs as well. The vertices of a Program-dependence graph represent individual statements in the code and the edges represent control and data flow between statements. Program-dependence graphs are typically used in program slicing (Korel and Laski, 1988) and optimization (Ferrante *et al.*, 1987).

Call graphs can be static or dynamic (Graham *et al.*, 1982). A static call graph (Allen, 1974) can be obtained by parsing the source code. Vertices of a static call graph represent all methods of a program and edges represents all possible method invocations. A dynamic call graph represents the execution of a certain scenario on a program. Unlike static call graphs, dynamic call graphs reflect the actual invocation structure of methods as edges. In this work, we deal with dynamic call graphs.

Dynamic call graphs can be represented as trees. Different tree representations exist to model dynamic call graphs. The following definitions are borrowed from (Washio *et al.*, 2005):

- A *free tree* is an undirected graph that is connected and acyclic.
- A *rooted unordered tree* is a directed acyclic graph satisfying: 1) there is a distinguished vertex called the root that has no entering edges; 2) every other vertex has exactly one entering edge; and 3) there is a unique path from the root to every other vertex.
- A *rooted ordered tree* is a rooted unordered tree that has a predefined ordering among each set of siblings. The order is implied by the left-to-right order in figures illustrating an ordered tree.
- An *unrooted unordered tree* is defined similar to a rooted unordered tree, except that no distinguished vertex exists.

In this thesis, we use rooted ordered trees to represent dynamic call graphs. In this representation the `main` method of the program is the root, the methods invoked directly by the `main` method are its children, and the siblings are ordered by execution time. More details are provided in Section 5.3. In general, dedicated tree mining algorithms can be less time consuming than general graph mining algorithms (Eichinger, 2011).

3.4.2 Call Graph Reduction

Dynamic call graphs tend to become very big and thus call-graph-based approaches do not scale well. Call-graph reduction is a technique to reduce the size of dynamic call-graphs. Different reduction techniques have been proposed, such as *total reduction* (where every distinct method is represented by exactly one node), *iteration reduction* (where iteratively executed structures caused by loops are reduced) and *recursion*

reduction (where recursive method calls are reduced). Eichinger (Eichinger, 2011) presents a survey of the reduction techniques as explained below. Figure 3.3 is also borrowed from Eichinger and illustrates unreduced call graphs (a), (h) and (j) and different reductions of those.

Total reduction probably provides the most efficient compression but loses the most information. There are three variants of total reduction. In *simple total reduction*, e.g., Figure 3.3(b), every distinct method is represented by exactly one node. When a method calls another method at least once in an execution, a directed edge connects the corresponding nodes. In *total reduction with edge weights*, e.g., Figure 3.3(c), every edge is annotated with a numerical weight which represents the total number of calls of the callee method from the caller method. In *total reduction with temporal edges*, e.g., Figure 3.3(d), temporal edges are introduced. A temporal edge is a directed edge that connects two methods which are executed consecutively and are invoked from the same method (Eichinger, 2011).

Reduction of Iterations compresses iteratively executed structures caused by loops. There are different variations of iteration reduction. In *unordered zero-one-many reduction*, e.g., Figure 3.3(e), graphs are rooted (unordered) trees where nodes represent methods and edges method invocations. In contrast to unreduced call graphs, an unordered zero-one-many-reduced graph ignores the order and omits isomorphic substructures which occur more than twice below the same parent node. *Ordered zero-one-many reduction*, e.g., Figure 3.3(f), omits only substructures which are executed more than twice in direct sequence. *Subtree reduction*, e.g., Figure 3.3(g), ignores the order and reduces subtrees executed iteratively by deleting all but one isomorphic subtree below the same parent node in an unreduced call tree. The edges

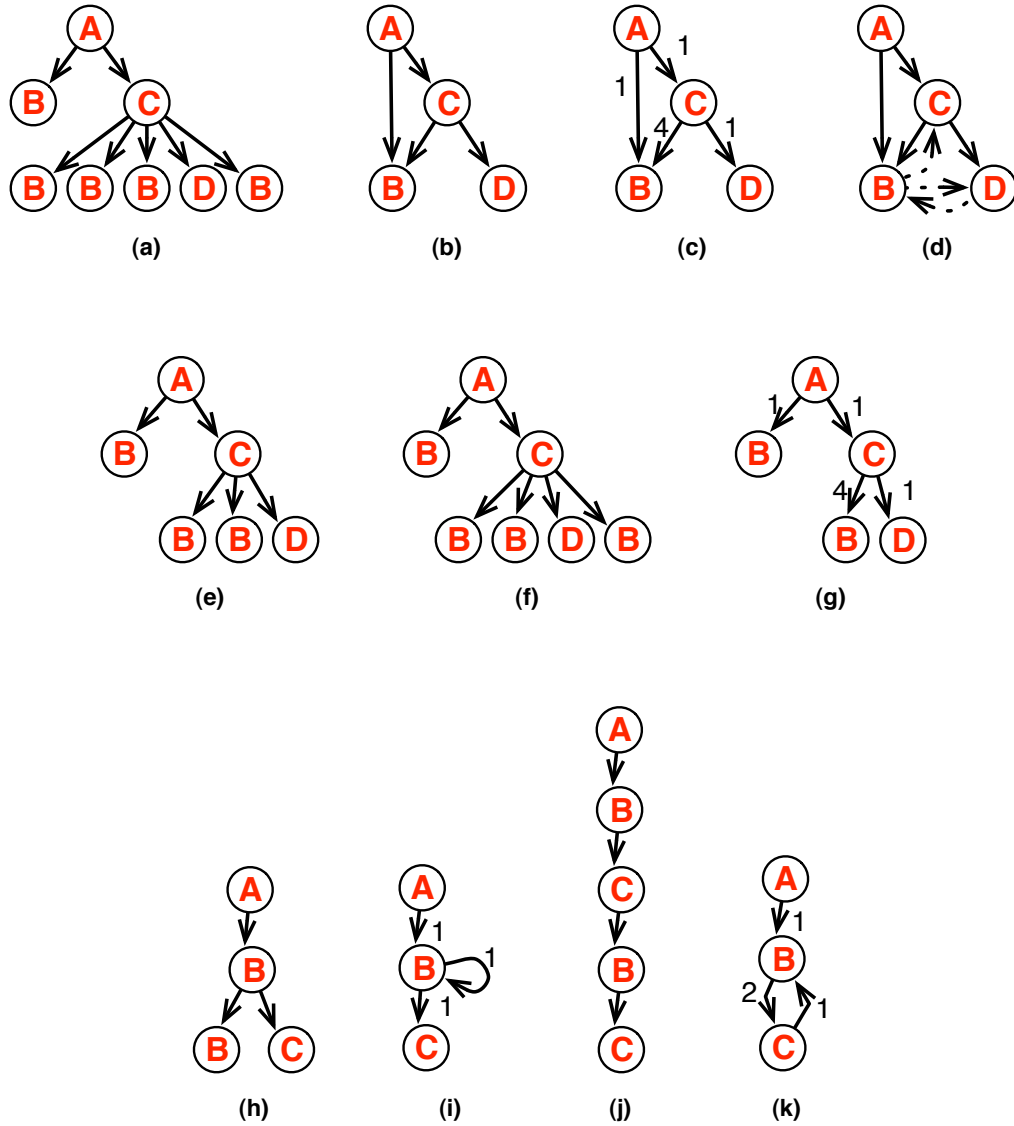


Figure 3.3: (a) An unreduced call graph (b) simple total reduction (c) total reduction with edge weights (d) total reduction with temporal edges (e) Unordered 0-1-m reduction (f) ordered 0-1-m reduction (g) subtree reduction (h) an unreduced call graph (i) direct reduction (j) an unreduced call graph (k) indirect reduction (Eichinger, 2011)

are weighted and numerical weights represent call frequencies (Eichinger, 2011).

Reduction of Recursions deals with recursive method calls. There are two variations. *Direct recursion* deals with the case where a method calls itself directly, e.g., method **b** in Figure 3.3(h). A possible reduction of this is presented in Figure 3.3(i) where a self loop at node **b** substitutes the recursion. In Figure 3.3(i), edge weights represent frequencies of iterations. *Indirect recursion* deals with the case where some method calls another method which in turn calls the first one again, which leads to a chain of arbitrary length of method calls. An example is provided in Figure 3.3(j) where **b** calls **c** which again calls **b**. Such indirect recursions can be reduced as shown in Figure 3.3(k) (Eichinger, 2011).

In this research we use a different reduction technique which extends the ordered zero-one-many by introducing a cascading mechanism which repeatedly reduces iterative substructures until none left. Figure 3.4 illustrates this approach. In this approach, graphs are rooted ordered trees where nodes represent methods and edges method invocations (Figure 3.4(a)). In this reduction we first omit isomorphic substructures which are executed more than once in direct sequence (Figure 3.4(b)), then we omit isomorphic chains of substructures which occur more than once in direct sequence (Figure 3.4(c)), then we apply this to the parents of the reduced nodes until finally reducing all iterations (Figure 3.4(d)). details of this reduction are provided in Section 5.3.

3.4.3 Mining Frequent Itemsets

In this section we define the problem of frequent-itemset mining. Our purpose is to introduce the concepts that have been exploited in subtree mining algorithms by

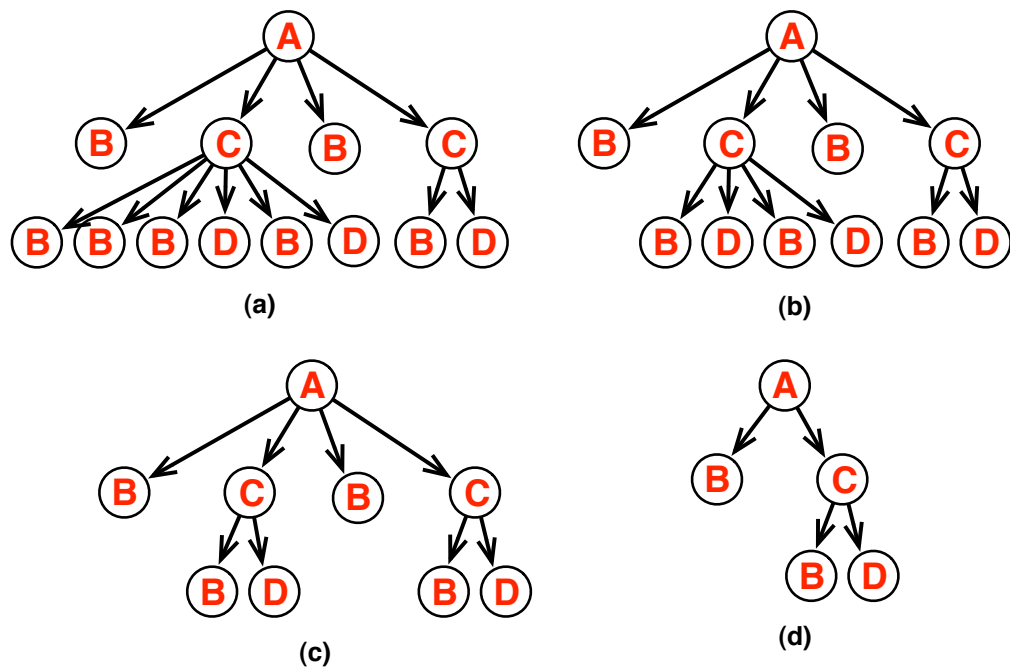


Figure 3.4: (a) An unreduced call graph (b) omitting isomorphic substructures in direct sequence (c) omitting isomorphic chain of substructures in direct sequence (d) repeating (b) and (c) for parent nodes until no iterations are left

discussing them first in the context of the simpler case of itemset mining.

Data mining is the process of finding interesting knowledge or patterns from databases (Hong *et al.*, 1999). Most of the available frequent-pattern mining algorithms borrow techniques from the domain of market-basket association rules mining (Chi *et al.*, 2004a), where algorithms have been devised to examine customer behavior with regards to the products they purchase.

In this thesis we follow the definition provided by Chi et al (Chi *et al.*, 2004a). They formally state the problem of frequent-itemset mining as follows. Given an alphabet Σ of items and a database \mathcal{D} of transactions $T \subseteq \Sigma$, we say that a transaction *supports* an itemset if the itemset is a subset of the transaction. The number of transactions in the database that support an itemset S is called the *frequency* of the itemset. The fraction of transactions that supports S is called the *support* of the itemset. Given a threshold *minimum support*, the frequent-itemset mining problem is to find the set $\mathcal{F} \subset 2^\Sigma$ of all itemsets S for which $support(S) \geq \text{minimum support}$. A well-known property of itemsets, which is the basis of frequent-itemset mining algorithms, is that $\forall S' \subseteq S : support(S') \geq support(S)$. In other words, all none-empty subsets of a frequent itemset must be also frequent. This property is called the *Apriori property* and is used to reduce the search space. As an example consider $\Sigma = \{A, B, C, D\}$, $\mathcal{D} = \{\{A, B, C\}, \{A, B, D\}, \{A, B, C, D\}\}$, and *minimum support* = 2/3, then:

$$F = \{\{A\}, \{B\}, \{C\}, \{D\}, \{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{A, B, C\}, \{A, B, D\}\}.$$

3.4.4 Mining Frequent Sequences

Sequence mining is a generalization of itemset mining where interesting rules among sequences of items are discovered. For example in the context of market-basket analysis, the extraction of which items are purchased after certain other items is one of the uses of sequence mining. More specifically, sequence mining is defined as the task of finding all sequences that are subsequence of at least a *minimum support* number of sequences in a set of transactions (Dong and Pei, 2007).

Sequence mining is used by researchers in the area of defect localization, e.g., (Hsu *et al.*, 2008), and feature location, e.g., (Sartipi and Safyallah, 2010). However, it is not the focus of this work (we use subtree mining). Therefore, we do not provide any further details here.

3.4.5 Mining Frequent Subtrees

Frequent-subtree mining is a special case of frequent-subgraph mining and is the basis of our defect localization technique in this thesis. Many algorithms have been proposed to discover frequent subtrees from a set of labeled trees. These algorithms vary in how they formulate the mining problem and details of their solutions (Chi *et al.*, 2004a). However, they have many similarities as well.

In this thesis we use the definition provided by Chi *et al.*, (Chi *et al.*, 2004a) for the problem of frequent-subtree mining. This problem is formally stated as follows . Given a threshold *minimum frequency*, a class of trees \mathcal{C} , a transitive subtree relation $P \preceq T$ between trees $P, T \in \mathcal{C}$, and a finite data set of trees $\mathcal{D} \subseteq \mathcal{C}$, the frequent-subtree mining problem is the problem of finding all trees $\mathcal{P} \subseteq \mathcal{C}$ such that no two trees in \mathcal{P} are isomorphic and for all $P \in \mathcal{P} : support(P, \mathcal{D}) = \sum_{T \in \mathcal{D}} d(P, T) \geq$

minimum frequency, where d is an anti-monotone function such that $\forall T \in \mathcal{C} : d(P', T) \geq d(P, T)$ if $P' \preceq P$. The subtree relation $P \preceq T$ defines whether a tree P occurs in a tree T . The simplest choice for function d is given by the indicator function:

$$d(P, T) = \begin{cases} 1 & \text{if } P \preceq T \\ 0 & \text{otherwise} \end{cases} .$$

In this simple case the frequency of a pattern tree is defined by the number of trees in the data set that contains the pattern tree. This matches the definition of frequency in itemset mining. Different kinds of the *contains* (subtree) relation are allowed in this definition, that are *bottom-up*, *induced* or *embedded* subtrees, as explained below (Chi *et al.*, 2004a):

- A **bottom-up subtree** for a rooted tree T (either ordered or unordered) can be obtained by taking a vertex v of T together with all v 's descendants and the corresponding edges. Note that nodes and edges of the bottom-up subtree preserve the original tree's properties such as labeling and sibling ordering.
- An **induced subtree** for a tree T (free, ordered or unordered) can be obtained by repeatedly removing leaf nodes (or possibly the root node if it has only one child) in T .
- An **embedded subtree** for a rooted unordered tree T is a subtree T' that saves the ancestor-descendant relationship which exists among vertices in T .

Figure 3.5 illustrates (a) a sample tree, (b) one of its bottom up subtrees, (c) one of its induced subtrees, and (d) one of its embedded subtrees.

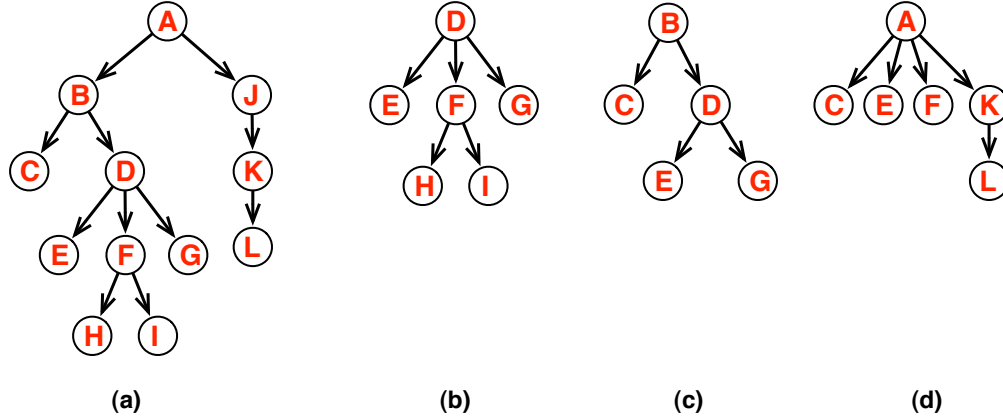


Figure 3.5: (a) Sample tree, (b) a bottom-up subtree, (c) an induced subtree, and (d) an embedded subtree

Since different definitions are possible for trees and subtrees, there are also different techniques available for mining them. Rooted ordered trees can be mined using FREQT algorithm (Asai *et al.*, 2002). Rooted unordered trees can be mined using HybridTreeMiner (Chi *et al.*, 2004b), Unot (Asai *et al.*, 2003), or FREQT (Nijssen and Kok, 2003) algorithms. Unrooted unordered trees can be mined using FreeTreeMiner (Chi *et al.*, 2003), HybridTreeMiner (Chi *et al.*, 2004b), or any arbitrary graph miners such as Gaston (Nijssen and Kok, 2004), as trees are special cases of graphs.

3.4.6 Approximate Tree Matching

Comparing trees has applications in many different areas (Shasha *et al.*, 1994). One common use of tree matching is to compare an unknown tree pattern against a number of template trees in order to assign the new pattern to the category to which the majority of its closest template trees belong (Duda and Hart, 1973).

Tree matching is a generalization of string matching, which is defined in two forms of exact and approximate. *Exact string matching* deals with two strings, a *subject string* and a *pattern string*. The problem is then defined as finding the pattern string in the subject string (Knuth *et al.*, 1977; Aho and Corasick, 1975). *Approximate string matching* is a little different. In this problem, we allow approximate matches, which are substrings that have some distance to the searched pattern and we are interested in minimizing this distance. This problem thus requires a distance metric called the *edit distance*. The most trivial is *Hamming distance*, which specifies the minimum number of substitutions required to change the pattern string into its corresponding substring in the subject string (Krcal, 2011). The most wide-spread is *Levenshtein distance*, which also considers deletion and insertion operations in addition to substitution (Krcal, 2011). *Damerau-Levenshtein distance* is another metric which generalizes Levenshtein distance by allowing transposition of two neighboring characters (Miller *et al.*, 2009).

Tree matching has also two variances. Similar to exact string matching, *Exact tree matching* is defined as searching in a tree (called the *subject tree*) for a specific subtree (called the *pattern tree*). *Approximate tree matching* is similar to *approximate string matching*. It is defined as finding subtrees of a subject tree with the least distance to a pattern tree. Edit distance of trees dates back to Lu (Lu, 1979), who first introduced *edit operations* on trees. The usual edit operations are:

- insertion: indicates that a node that exists in the pattern tree is missing from its corresponding subtree in the subject tree.
- deletion: indicates that a node that does not exist in the pattern tree is observed its corresponding subtree in the subject tree.

- substitution: indicates that a node that exists in the pattern tree is substituted with a different node in its corresponding subtree in the subject tree

Figure 3.6 illustrates a pattern and its approximate matches, where (a) presents the pattern, (b) presents the case of insertion (subtree rooted at **C** is missing), (c) illustrates a case of deletion (vertex **H** is new), and (d) presents a substitution (subtree rooted at **C** is substituted by subtree rooted at **I**).

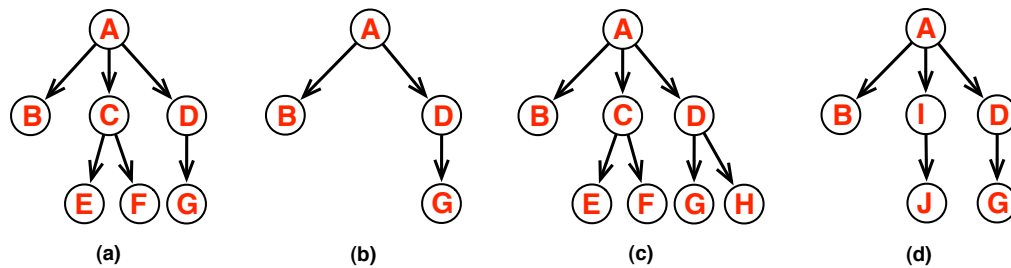


Figure 3.6: A pattern and approximate matches in a call tree: (a) pattern (b) insertion (c) deletion (d) substitution

Note that in both string and tree matching, we allow the pattern to match only part of the subject. In approximate pattern matching, we also allow the pattern to suppress the details of the subject in which we are not interested.

Chapter 4

Defect Localization Literature

When the execution of a program fails, the programmer often analyzes a memory dump, in case the program crashed and forensic data may help to indicate why and where the program failed. In another approach, the programmer inserts break-points or print-statements around the code that is thought to be the cause of the failure, to examine the program state (i.e., variable values) in order to find hints leading to the location of defects. This approach places the burden on programmers to decide which code segments are likely to have caused the failure, as well as which variable values should be monitored. In this case, the programmer must develop a strategy to examine only meaningful information.

There is a high demand for automatic defect localization techniques that can guide programmers to the location of defects with minimal human intervention. As Wong and Debroy discuss in (Wong and Debroy, 2010), this demand has led to the proposal and development of various techniques over the years. These techniques pursue quite similar goals. However, their underlying ideas can be different from one another, since these ideas stem from several different disciplines. No article (regardless of breadth

or depth) can hope to cover all defect localization techniques.

There are several ways to classify defect localization techniques. One way of doing this is to categorize them into *static* and *dynamic* approaches. In static approaches the program's source code, documentation, or graph representations of the source code (such as static dependence graph) are analyzed to localize defects. Static approaches consider potential relations that may never happen in any executions. Therefore, they tend to have many false positives. In contrast, dynamic approaches consider program execution information such as run time program state, or execution traces. Dynamic approaches have considerably fewer false positives but can have poor coverage of the source code. The proposed approach in this thesis is a dynamic one. Therefore in the following section, we focus on dynamic approaches and discuss them in more detail.

4.1 Dynamic Approaches

The essence of dynamic approaches is examining execution information in passing and failing executions. Such execution information includes statements executed, branches taken, methods called, values of variables, etc. Dynamic approaches can further be categorized as follows.

4.1.1 Delta Debugging Based

Delta debugging was first introduced by Zeller (Zeller, 1999). The idea behind delta debugging is to isolate *failure-inducing entities*, where the term “entity” can refer to program input (Zeller and Hildebrandt, 2002), variables (Zeller, 2002), code (Zeller, 1999), etc. In a defect localization scenario, one may want to highlight within the

set of program variables, those variable-value pairs that are relevant to the failure (i.e., suspicious variables). Then, track causal relations between variables to identify variables affecting the values of suspicious variables, in order to ultimately find the origin of the failure (i.e., the defect) (Cleve and Zeller, 2005; Zeller, 2002). To identify suspicious variables, a delta debugging based approach first compares values of variables in a failing execution with their respective values in a passing execution to identify the changes, which are called *failure-inducing changes*. Then, it follows an experimental narrowing process to find a minimal subset of the changes. An example of such a narrowing process is that we replace a suspicious variable's value in a successful test with its corresponding value in a failed test, to see if it makes the previously successful test fail, and keep it as failure-inducing if so.

The delta debugging idea has also been applied to isolate failure-inducing user interactions, thread schedules, or code changes. For example, Zeller (Zeller, 1999) searches for minimal failure-inducing code changes by examining different versions of a program from a version-control system. Wang and Roychoudhury (Wang and Roychoudhury, 2005) present a technique that automatically analyzes the execution path of a failed test, and alters the outcome of branches in that path to produce a successful execution. The branch statements whose outcomes have been changed are recorded as failure-inducing.

Delta debugging is in essence a way of finding a starting point to search for the root cause of a failure. This starting point may be a failure-inducing input, failure-inducing variable assignments or suspicious code changes that have happened since the last time a test has run successfully. The proposed approach in this thesis also seeks a starting point for root cause analysis. However, we are using dynamic method

call relations and their changes from passing to failing executions to find suspicious method calls. In general, analyzing the program state space and designing extra tests to narrow down failure-inducing changes (as performed in delta debugging) is more costly than examining call trees (as performed in our technique). In this sense delta debugging approaches are not appropriate for large systems.

4.1.2 Program Slicing Based

In program slicing-based defect localization, the goal is to narrow down the debugging search space via *slicing* and *dicing*. This idea has been applied to *static* (Weiser, 1982), *dynamic* (Agrawal *et al.*, 1993a; Sterling and Olsson, 2005; Zhang *et al.*, 2005), and *execution* (Wong and Qi, 2006) slices. In this context, a *static slice* is the set of statements of a program which might affect the value of a particular output (or the value of a variable instance). A *dynamic slice* is the set of statements of a program which do affect the value of the output (or the value of a variable instance) on the execution of a particular input. An *execution slice* is the set of a program's basic blocks or a program's decisions executed by a test input.

The underlying assumption in static/dynamic slicing-based defect localization is that if a test case fails due to an incorrect variable value at a statement, then the defect should be found in the static/dynamic slice associated with that statement. Therefore, the attention is focused on that slice and the rest of the program is ignored in searching for the defect.

Lyle and Weiser (Lyle and Weiser, 1987) proposed the notion of program dicing which attempts to further reduce the search domain for possible locations of a defect. In this approach, the search for the defect is narrowed down by considering the slice

that belongs to a failing execution minus that of a passing execution, i.e., a *dice*.

Saha et al., (Saha *et al.*, 2011) applied dynamic slicing to analyze data-centric programs, where one or more entries in an output collection may be faulty. In this approach, they break the execution trace into multiple slices, such that each slice maps to an entry in the output collection. Then, they compute the semantic difference between the slice that corresponds to a correct entry and the one that corresponds to an incorrect entry to identify potentially faulty statements.

Like static and dynamic dicing, the underlying assumption behind execution dicing is that if a test case fails because of a defect in the code, the defect should be found in the execution slice of the failing test case minus the statements in a passing test case. Agrawal et al., (Agrawal *et al.*, 1995) proposed an execution slicing-based defect localization technique where they formed all possible dices in which a successful slice is subtracted from a failed slice, and then presented a randomly chosen dice to the programmer as an initial guess at the defect's location.

Gupta et al., (Gupta *et al.*, 2005) integrated the potential of delta debugging algorithm with the benefit of dynamic program slicing to narrow down the search for defective code. In this approach, they use a delta debugging algorithm to identify minimal failure-inducing input. Then, they use the minimal failure-inducing input to compute a forward dynamic slice. Next, they intersect the statements in this forward dynamic slice with the statements in the backward dynamic slice of the erroneous output. The intersected statements compose what is called a *failure-inducing chop*. In this process, the forward dynamic slice of a variable at a point in the execution trace includes all those executed statements that are affected by the value of the variable at that point. In contrast, the backward dynamic slice of a variable at a

point in the execution trace includes all those executed statements which affect the value of the variable at that point. The failure-inducing chops contain statements that are likely to have caused the failure. They are expected to be much smaller than backward dynamic slices since they capture only those statements of the dynamic slices that are affected by the minimal failure-inducing input.

As we already mentioned, the basic idea behind our defect localization technique is to note the differences between dynamic call trees associated with passing executions and that of a failing execution. This idea is somehow similar to the idea behind slicing and dicing. In this regard, the dynamic method call tree associated with a failing execution can be regarded as a slice to search for the defect. Similar to dicing, we assume that method calls from the dynamic call tree of a faulty execution that are not executed in passing executions are more likely to point to defects. However, we do not stop at this point and consider other cases too. For example, we assume that method calls appearing in passing executions only and method calls shared between passing and failing executions can also point to the defect.

In terms of dicing, we assume one failing and more than one passing tests which is different from simple dicing (where difference of a failing and a passing slice is reported). To handle many passing tests Agrawal et al., (Agrawal *et al.*, 1995) performed a one-to-one dicing and randomly presented the dices to the user. Also Agrawal (Agrawal, 1992) proposed that one can deduce the union of passing slices from the intersection of all failing slices to create a dice. Based on experimental evaluation, Renieris and Reiss (Renieris and Reiss, 2003) claim that other approaches such as the nearest neighbor (discussed in the upcoming subsection) work better than union and intersection models. In our technique, we perform a one-to-one comparison

between the call tree of the failing execution and a select set of passing executions and propose a ranking mechanism to present the suspicious differences to the user.

Moreover, static/dynamic slicing can be a complementary approach to ours. Static/Dynamic slicing requires a starting point which our approach can provide. After finding a starting method using our technique, one could use static/dynamic slices to find the root cause of a failure.

4.1.3 Program Spectra Based

Program spectra-based approaches rely on the *spectrum* of a program to localize defects in the code. The term “spectrum” was first introduced by Reps et al., (Reps *et al.*, 1997), and refers to a record of a program’s execution in certain aspects such as how statements and conditional branches are executed with respect to each test, the number of times each line of the program is executed, function call counts, program paths, program slices, and use-def chains.

When an execution fails, a program spectrum can be utilized to identify suspicious code. Different program spectra-based approaches have been proposed. In most of the approaches, the spectra of a program in passing and failing executions is used to evaluate certain suspiciousness measures for every program construct. The suspiciousness numbers are then used to rank program constructs, such that constructs with greatest likelihood of being related to the defect rank the highest. Table 4.1.3 lists a number of suspiciousness formulations from the literature. We use the notation $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$ provided by Naish et al., (Naish *et al.*, 2011) in this table. In this notation, the first part of the subscript indicates whether the statement was executed (e) or not (n) and the second indicates whether the test passed (p) or failed

(*f*). For example, a_{ep} of a statement is the number of tests passed and executed the statement.

Table 4.1: Suspiciousness formulations presented in literature

Name	Formula
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$
Optimal Ranking Formula	$\begin{cases} -1 & \text{if } a_{nf} > 0 \\ a_{np} & \text{otherwise} \end{cases}$

Spectra-based approaches differ on many levels including: the type of information (i.e. spectra) they collect (such as statement hit, function call frequency, etc); granularity of suspiciousness formulations (e.g., statement, basic block of statements, method, class, component); the type of analysis (statement coverage analysis, sequence analysis, call graph mining); etc.

Tarantula (Jones and Harrold, 2005) is one of the pioneers of spectra-based defect localization. In Tarantula (Jones and Harrold, 2005), a program spectrum includes the statements that each test covers in its execution. The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be defective than those that are primarily executed by passed test cases. Based on this intuition, a suspiciousness metric is devised which expresses the likelihood of a statement being defective (See Table 4.1.3). The suspiciousness numbers are computed for all statements in the code and visualized using a series of colors. In this visualization, statements that are executed primarily by failed test cases and are thus, highly suspicious of being defective, are colored red to denote

“danger”; statements that are executed primarily by passed test cases and are thus, not likely to be defective, are colored green to denote “safety”; and statements that are executed by a mixture of passed and failed test cases and thus do not lend themselves to suspicion or safety, are colored yellow to denote “caution”.

The literature contains many metrics that can be used for ranking statements according to how likely they are to be defective. Naish et al., (Naish *et al.*, 2011) presented a very simple program with a single defect to model the defect localization problem and evaluate the effectiveness of different proposed metrics which is followed by devising an optimal ranking method (Table 4.1.3). Wong et al., (Wong *et al.*, 2010) introduced and evaluated a number of coverage-based heuristics to localize faults. Abreu et al., (Abreu *et al.*, 2009a) aimed at improving Tarantula (Jones and Harrold, 2005) by evaluating different scoring functions besides Tarantula within the same framework.

Renieris and Reiss (Renieris and Reiss, 2003) introduced the idea of *nearest neighborhood*. Their method assumes the existence of a faulty run and a larger number of passing runs. It then selects according to a distance criterion the passing run that most resembles the failing run, compares the spectra corresponding to these two runs, and produces a report of “suspicious” parts of the program. In this approach, the program spectrum includes information about basic blocks executed and their execution counts. To experimentally validate the viability of their method, they implemented it in a tool, WHITHER, using basic block profiling spectra.

Pinpoint (Chen *et al.*, 2002) is a framework for root cause analysis on the J2EE platform and is targeted at large, dynamic Internet services, such as web-mail services and search engines. In this approach, they dynamically trace real client requests as

they travel through a system. For each request, they record its believed success or failure, and the set of components used to service it. Then they use a data clustering algorithm to correlate the failures of requests to the components most likely to have caused them. In this clustering, similarity is calculated based on how often components are and are not used together in requests that are highly correlated with failures of requests. Finally, they report components whose occurrences are most correlated with failures, and hence where the root cause is likely to lie.

Wong et al., (Wong *et al.*, 2012b) presented a crosstab-based statistical technique to localize defects. The presented approach makes use of the coverage information of executable statements and the execution result (success or failure) associated with each test case to assess the degree of association between execution result and the coverage of each statement by means of statistical coefficients.

Wong et al., (Wong *et al.*, 2012a) proposed DStar, a technique which uses statement coverage information for fault localization. In this technique the suspiciousness of program statements are evaluated using a similarity coefficient. This basic idea is that the closer the execution pattern (i.e., the coverage vector) of a statement is to the failure pattern (result vector) of the test cases, the more likely the statement is to be faulty, and consequently the more suspicious the statement seems. Thus, similarity measures or coefficients can be used to quantify this closeness. The degree of similarity can be interpreted as the suspiciousness of the statements.

Most existing defect localization techniques focus on identifying and reporting single code constructs (e.g., statement, method, etc) that may contain a defect. Even in cases where a defect involves a single code construct, it is generally hard to understand the defect by looking at that construct in isolation. Defects typically manifest

themselves in a specific context, and knowing that context is necessary to diagnose and correct the Defect. Hsu et al., (Hsu *et al.*, 2008) extends Tarantula’s idea by incorporating sequential patterns as context. They use Tarantula’s suspiciousness formulation to compute suspiciousness of statements. Then, they filter less suspicious statements (suspiciousness value of less than 0.6, intuitively representing statements with less than 60% chance of being related to the failure). Next, they mine frequent longest common subsequences from the filtered failing traces to identify bug signatures. A longest common subsequence is a common subsequence that is not contained in any other common subsequence. Such patterns present some kind of defect context to the user.

Dallmeier et al., (Dallmeier *et al.*, 2005) proposed a tool for identifying defective classes in Java applications. This tool instruments a given Java program such that sequences of method calls are collected on a per-object basis. Then, it computes a suspiciousness weight for each sequence. They proposed separate weight functions for the cases (1) where one passing and one failing executions are available:

$$w(p) = \begin{cases} 1 & \text{if } p \notin P_{pass} \cap P_{fail} \text{ - that is, } p \text{ is new or missing} \\ 0 & \text{otherwise} \end{cases},$$

as well as (2) when multiple passing and one failing executions exist:

$$w(p) = \begin{cases} Support(p) & \text{if } p \notin P_{fail} \\ 1 - Support(p) & \text{if } p \in P_{fail} \end{cases},$$

where w is a weight assigned to a sequence of method calls p based on $Support(p)$,

the percentage of passing traces exhibiting sequence p , for sequences that do or do not belong to P_{fail} , the set of sequences in the failing run. Next, it computes the average sequence weight of a class, from the weights of all sequences. For case (1) it uses:

$$avg = \frac{1}{t} \sum_{p \in P_{pass} \cup P_{fail}} w(p) \text{ where } t = |P_{pass} \cup P_{fail}|,$$

and for case (2) it uses:

$$avg = \frac{1}{t} \sum_{p \in \bigcup P(r_i)} w(p) \text{ where } t = \left| \bigcup_{i=0}^n P(r_i) \right|,$$

to compute the average weight. In this formulation r_0 stands for the failing run and $r_i, i = 1, ..$ stands for the passing runs. They claim that classes with the highest average weight are most likely to contain a defect.

Di Fatta et al., (Di Fatta *et al.*, 2006) proposed a call graph based defect localization approach. In this approach, they collect method call traces and represent them as rooted ordered trees. Then, reduce the iterations from the trees and mine frequent subtrees from the reduced trees. The call trees analyzed are large and lead to scalability problems. Hence, they limit the size of the subtrees mined to a maximum of four nodes. Then they provide suspiciousness numbers for all methods in the code. The suspiciousness formula is based on the support of a method in two sets of trees: 1) subtrees contained in call graphs of failing executions which are not frequent in call graphs of passing executions, also called the *specific neighborhood (SN)*; 2) the set of call graphs of passing executions, D_{pass} . Suspiciousness of method m is computed as:

$$P(m) = \frac{support(g_m, SN)}{support(g_m, SN) + support(g_m, D_{pass})},$$

where g_m denotes all graphs containing method m and $support(g, D)$ denotes the support of a graph g , i.e., the fraction of graphs in a graph database D containing

g. Note that this formulation assigns zero to methods which do not occur within SN and one to methods which occur in SN but not in passing program executions D_{pass} . The intuition is that methods that appear in all failing executions and not frequently seen in passing executions are more likely to be defective. Although Di Fatta et al., do not specifically discuss the types of defects they can handle, they appear to be able to localize structure-affecting defects.

Liu et al., (Liu *et al.*, 2005) also proposed a call graph based approach to deal with structure-affecting defects. In this approach, they mine sub graphs from reduced call graphs associated with passing and failing executions. Then, assign binary vectors for each execution where every element in the vector indicates if a certain subgraph is included in the corresponding call graph for that execution. Using these vectors, a support-vector machine (SVM) classifier (Vapnik, 1995) is learned which decides if a program execution is passing or failing. For every method two classifiers are learned: one based on call graphs including the respective method and one based on graphs without this method. If the precision rises significantly when adding graphs containing a certain method, this method is deemed more likely to contain a defect. This approach does not provide any ranking of the defect-relevant method set.

Cheng et al., (Cheng *et al.*, 2009) also proposed a call graph based approach to deal with structure-affecting defects. They first used the LEAP algorithm (Yan *et al.*, 2008) to mine the top-k most discriminative subgraphs, i.e., subgraphs contrasting failing executions from passing ones and thus are assumed to have an increased likelihood to be related to defects. Then, they present them to the user. In this approach, no method ranking is provided. Cheng et al., also applied their approach on finer-grained basic-block-level call graphs.

Eichinger et al., (Eichinger *et al.*, 2008) also used graph mining for defect localization for both structure and frequency affecting defects. In this approach, they mine reduced call graphs and partition the resulting frequent subgraphs into two sets: 1) the set of subgraphs which occur in both passing and failing executions; 2) the set of subgraphs which only occur in failing executions. They use the former to provide an *Information Gain*-based score $P_e(m)$ which identifies which edge weights of call graphs from a program are most significant to discriminate between passing and failing. *Information Gain* (Quinlan, 1993) is a measure based on entropy. It is defined in the interval $[0, 1]$ and quantifies the ability of an attribute A to discriminate between classes in a dataset (without a restriction to binary classes). In the context of this work, the data set is the set of passing and failing tests, attribute A represents a specific method call such as a call from method a to method b and values of this attribute correspond to frequencies of this call in different tests. The aim of this is to see which method call can best discriminate between passing and failing tests. Eichinger et al., use subgraphs only occurring in failing executions to provide a structural score:

$$P_s(m) = \text{support}(m, SG_{fail}).$$

This score is based on the support of method m in the the set of subgraphs only occurring in failing executions. This is done because *Information gain*-based scoring cannot detect defects that are not call frequency affecting. Also, it does not consider subgraphs which occur in failing executions only (as some defects can result in such subgraphs). Finally, they normalize and combine (add) the two scores to compute suspiciousness of methods and provide a ranking:

$$P(m) = \frac{P_e(m)}{2 \max_{n \in T} (P_e(n))} + \frac{P_s(m)}{2 \max_{n \in T} (P_s(n))},$$

where n is a method in a program trace t in the collection of all traces T .

In a follow-up of their work, Eichinger et al., (Eichinger *et al.*, 2010b) extended their approach to support dataflow affecting defects. In this approach, they mine dataflow enabled call graphs, and, similar to their previous work (Eichinger *et al.*, 2008), assign an *Information Gain Ratio*-based and a structure-based suspiciousness score to the methods. *Information Gain Ratio* is a measure based on *Information Gain* which similarly measures the discriminativeness of an attribute A when values $v \in A$ partition the dataset D . The *Information Gain Ratio* normalizes the *Information Gain* value by *SplitInfo*, which is the entropy of the discretization of the attribute into intervals. Eichinger et al., also extended their work to deal with concurrency defects in (Eichinger *et al.*, 2010a). In (Eichinger *et al.*, 2011) they provided a hierarchical approach to localize defects, which analyzes graphs of a coarse granularity before it zooms-in into finer-grained graphs.

As Eichinger et al., (Eichinger, 2011) explains, mining dynamic call graphs is a relatively recent and very promising approach in the dynamic defect-localization literature. Our aim in this thesis is not to develop a technique which rules out any existing alternatives, but to investigate yet another usage of call graphs for defect localization.

4.1.4 Discussion of Spectra-Based Approaches

The Spectra-based techniques we overviewed in the previous subsection have positive and negative characteristics. In this section we discuss strengths and weaknesses of the reviewed techniques. We also identify how these techniques relate to the proposed defect localization technique in this thesis. Table 4.2 provides a list of related spectra-based approaches and their characteristics.

Table 4.2: Related spectra-based approaches

Approach	Spectra	Assumption	Suspiciousness Formulation
Reps et al. (Reps <i>et al.</i> , 1997)	Number of times each different loop-free path is executed	m pass, n fail (m,n>1)	NA
Jones and Harrold (Jones and Harrold, 2005)	Statement hit	m pass, n fail (m,n>1)	Tarantula
Naish et al. (Naish <i>et al.</i> , 2011)	Statement hit	m pass, n fail (m,n>1)	Optimal ranking metric
Abreu et al. (Abreu <i>et al.</i> , 2009a)	Statement hit	m pass, n fail (m,n>1)	Ochiai coefficient
Renieris and Reiss (Renieris and Reiss, 2003)	Block hit	m pass, 1 fail (m>1)	NA
Chen et al. (Chen <i>et al.</i> , 2002)	Component hit	m pass, n fail (m,n>1)	NA
Hsu et al. (Hsu <i>et al.</i> , 2008)	Statement hit	m pass, n fail (m,n>1)	Tarantula
Dallmeier et al. (Dallmeier <i>et al.</i> , 2005)	Method hit	m pass, 1 fail or m pass, n fail (m,n>1)	Dallmeier
Di Fatta et al. (Di Fatta <i>et al.</i> , 2006)	Call graph	m pass, n fail (m,n>1)	Di Fatta
Continued on next page			

Table 4.2 – continued from previous page

Approach	Spectra	Assumption	Suspiciousness Formulation
Liu et al. (Liu <i>et al.</i> , 2005)	Call graph	m pass, n fail (m,n>1)	NA
Cheng et al. (Cheng <i>et al.</i> , 2009)	Call graph	m pass, n fail (m,n>1)	Information gain
Eichinger et al. (Eichinger <i>et al.</i> , 2008)	Call graph	m pass, n fail (m,n>1)	Information gain and support in frequent failing subgraphs
Eichinger et al. (Eichinger <i>et al.</i> , 2010b)	Call graph	m pass, n fail (m,n>1)	Information gain ratio, support in frequent failing subgraphs and support in frequent correct subgraphs
Eichinger et al. (Eichinger <i>et al.</i> , 2010a)	Call graph	m pass, n fail (m,n>1)	Information gain and information gain ratio
Eichinger et al. (Eichinger <i>et al.</i> , 2011)	Call graph	m pass, n fail (m,n>1)	Information gain
OUR APPROACH	Call graph	m pass, 1 fail (m>1)	An extension of Dallmeier's formulation, <i>parentDiffSize</i> and <i>diffSize</i>

Among all spectra-based approaches, statement-based approaches such as (Reps *et al.*, 1997; Jones and Harrold, 2005; Naish *et al.*, 2011; Abreu *et al.*, 2009a; Hsu

et al., 2008; Renieris and Reiss, 2003) locate defects more precisely by providing finer-grained results. However, they need to deal with very long execution traces and thus it is not clear in this stage whether they can handle large applications effectively. More granular approaches such as call-graph-based techniques (including our approach) are more extensible in this sense.

Machine learning based approaches such as (Chen *et al.*, 2002; Liu *et al.*, 2005) require a non-trivial split of the test cases, where the number of passing and failing cases are about the same, to train their machines. Therefore, they are not suited for failures that happen very rarely. Specifically, they are not good candidates to handle the case of one failing and many passing tests that we target in this thesis.

Eichinger *et al.*, (Eichinger *et al.*, 2010a; Eichinger, 2011) tried a number of different feature selection algorithms including *Information Gain* and *Information Gain Ratio* for defect localization. Using some preliminary experiments they evaluated a number of such techniques with the result that those based on entropy are best suited for defect localization. They identified that *Information Gain* produces the best results when applied to a small Diff tool taken from (Darwin, 2004). However, *Information Gain* does not provide good suspiciousness numbers for a test suite with imbalanced class distributions (different number of passing and failing test cases) (Eichinger, 2011). For example, referring to Figure 4.1, assume that we have two classes “pass” and “fail”, where we have 99 cases pass and one fails and in all passes, method **a** calls method **b**, while in the failing case this method call does not happen. In this case, *Information Gain* of this method call is about 0.08. Thus, the suspiciousness of this method call and hence that of method **a** is 8%, which does not make sense since a missing call that has always been present is very suspicious.

The same is true if in all passing executions, method **a** calls **b** 10 times, while in the failing case this call happens only once. This reason behind this problem is that, in an imbalanced class distribution the entropy of the classification of the cases is low.

Test	$a \rightarrow b$...	Class
t1	1	...	pass
t2	1	...	pass
t3	1	...	pass
⋮	⋮	⋮	⋮
t100	0	...	fail

Figure 4.1: An imbalanced class distribution where 99 out of 100 tests pass and only one fails

Compared with *Information Gain*, *Information Gain Ratio* is robust regarding imbalanced class distributions because it normalizes the *Information Gain* by its *SplitInfo* (Quinlan, 1993). Revisiting our previous example, the *Information Gain Ratio* in this case is one, which is closer our expectation. However, *Information Gain Ratio* is unstable if the split is near-trivial (i.e., almost all cases pass or almost all fail) (Quinlan, 1993). To avoid this undesirable eventuality, any test suite used must contain more than a minimum number of passing and failing cases. Therefore, approaches such as (Cheng *et al.*, 2009; Eichinger *et al.*, 2008, 2011, 2010a) that use *Information Gain* and (Eichinger *et al.*, 2010b,a) that use *Information Gain Ratio* are not suitable for the case of one failing test and many passing tests that we target in this thesis.

Approaches such as (Di Fatta *et al.*, 2006; Dallmeier *et al.*, 2005; Eichinger *et al.*, 2008, 2010b) incorporate a support-based formulation to compute suspiciousness values. Support-based formulations can not handle stochastic behavior very well and

may create more false positives or false negatives. Assume in a passing execution method **a** can call any of the methods **b**, **c** or **d**. Assume that the support of “**a** calling **b**” is 2% in passing cases. In this case, if the failing case also exhibits “**a** calling **b**”, then according to a support-based formulation such as Dallmeier’s suspiciousness formula (Dallmeier *et al.*, 2005) the suspiciousness of this call is 98%. This can be an example of a false positive. “**a** calling **b**” can reflect a correct call which has also been exhibited in a failing test case. However, if the failing case does not exhibit “**a** calling **b**” then suspiciousness of this call is 2%. This can be an example of a false negative. In this case, “**a** calling **b**” can be an expected call that is missing in the failing test case. In our approach, we mitigate this effect by incorporating several other analysis on call relations to find “expected” behavior. In this context if “**a** calling **b**” is rare but belongs to a part of the call tree that is specific to the failing feature, it is ranked higher in the list of suspicious methods.

Another issue with Dallmeier *et al.*, (Dallmeier *et al.*, 2005) is that it has the granularity of a class which is less informative compared with method-level approaches. For Di Fatta *et al.*, (Di Fatta *et al.*, 2006), another issue is that it gives zero for methods that do not occur in the specific neighborhood, implying that they are not suspicious. This, for example, happens for method calls that are frequent in passing executions but not frequent in failing executions. As we will see later in this discussion, such method calls can also point to defects in the code. Eichinger *et al.*, (Eichinger *et al.*, 2008, 2010b) use *InformationGain*-based formulation for frequency-affecting defects and support-based formulations for structure-affecting defects. They accordingly inherit the weaknesses and strength of both approaches.

Unlike our approach, the call-graph-based approaches we studied in the previous

section are not suited to handle failures that happen very rarely. In a case in which a defect is very unlikely to reveal itself, there may be very few failing cases (sometimes only one case) that represent this defect. This can lead to two possible outcomes:

- If we assume a single defect in the code, this means having very few failing cases as compared with passing ones which most approaches cannot handle very well.
- If we assume multiple defects in the code, then only a small percentage of failed tests represents the rare failure. Assume that execution of method m is not expected in a correct execution and is related to the rare defect. Since the defect affects a small portion of the failed tests, m is observed in only a small percentage of the failed tests. This leads to a low suspiciousness value for m .

Spectra-based approaches have another deficiency as well. They aggregate potentially irrelevant information to compute suspiciousness measures. This can be considered from different aspects:

- *Aggregating information about different functionalities exercised in a test suite:* The following cases are considered. i) If a certain method call contributes to the execution of multiple functionalities, the total frequency of this call in a test case does not convey meaningful information about either of the functionalities. In this case, changes in the frequency of the method call from passing to failing executions does not provide useful insight into the failure. Using such an aggregation could increase both false positives and false negatives in defect localization. Availability of loops in programs can further aggravate this effect; ii) Assume that among other functionalities of a system f_1 and f_2 are exercised in a test suite. Assume that whenever performed, f_1 is always successful, but

f_2 fails in a portion of runs (and we are searching for the method related to the failure of f_2). If by chance the test suite exercises functionality f_1 only in the failing tests, and if method m_1 is specific to functionality f_1 , then we only observe m_1 in failing tests. In an spectra-based approach such a method gets a high suspiciousness value and consequently considered defect-related, although it is not (i.e., false positive). Knowing more about the exercised functionalities in addition to the outcome of test executions can help mitigate this problem.

- *Aggregating information about different failure sources*: Failures raised in a test suite may have different reasons. They may happen as a result of distinct defects in the code and reveal themselves via change in the frequencies of different method calls. For example the failing test cases of a test suite can represent failures of two functionalities f_1 and f_2 with different underlying reasons. In such a case, it only makes sense to have different classifications of passing and failing test cases for each failing functionality. Considering the functionalities exercised in tests, especially the failing functionality, and by performing *Feature Location*, we could mitigate this risk quite effectively.

Both statement-based and some of the call-graph-based approaches (Di Fatta *et al.*, 2006; Eichinger *et al.*, 2008) assign high suspiciousness values to entities which have been seen mainly in failing test cases. They assume that the more successful tests execute a certain piece of code, the less likely it is that this code is defective. Similarly, the more failed tests execute a certain piece of code, the more likely it is that this code is defective. We believe that a missing code entity can lead us to a defect in a related entity. For example assume that in all passing cases method **a** calls method **b** but in all failing test cases the call to method **b** is missing. In such a

case it is probable that method **a** has a defect (e.g., an exception, or change in the path that is executed inside method **a**) that prevents it from calling method **b**. In our approach we also consider an entity that is missing from the failing call to be suspicious. Note that in some cases “a missing entity” may be our only clue to the location of the defect.

Various techniques and tools provided in this domain consider specific defect types in specific programming languages and under certain assumptions. Studies comparing different defect localization approaches (Rutar *et al.*, 2004; Santelices *et al.*, 2009) came to the conclusion that no single approach strictly outperforms any other for all kinds of defects or can detect all kinds of defects. Therefore, a combination of techniques should be used to build a powerful defect localization tool.

4.1.5 Other Approaches

There are many other dynamic approaches for defect localization which are quite different from our approach. Two examples are *statistical defect localization* and *defect localization with graphical models*. In *statistical defect localization* (Liblit *et al.*, 2005) programs are first instrumented to collect statistics about run time evaluations of certain predicates (e.g., evaluations of conditional statements and function return values). Take the predicate “ $idx < LENGTH$ ” for example, where variable idx is an index into a buffer of length $LENGTH$. This predicate checks whether accesses to the buffer ever exceed the upper bound. Statistics on the evaluations of predicates are collected over multiple executions and analyzed afterwards to localize defects. For example one can calculate the likelihood that the evaluation of a predicate to true correlates with failing and use this to find suspicious code segments. In *defect*

localization with graphical models (Dietz *et al.*, 2009) graphical models are used to localize defects. Graphical models are introduced in machine-learning and bring together concepts from graph theory and probability theory (Jordan, 1999). Well-known representatives of graphical models are Bayesian networks, which are also known as directed acyclic graphical models or belief networks (Jensen, 2009).

Chapter 5

Defect Localization using Dynamic Call Tree Mining and Matching

In this chapter we discuss our defect localization technique. We first present an overview of the proposed technique in Section 5.1, followed by a detailed discussion of the defect localization process in Sections 5.2 to 5.6. We conclude with a discussion of the benefits and shortcomings of the proposed technique in Section 5.7. The proposed technique has also been published in (Yousefi and Wassyng, 2013) and (Yousefi and Sartipi, 2011).

5.1 Overview

In this section, we provide an overview of the proposed defect localization technique. We first explain the problem we are targeting. Then, we briefly present the idea behind our approach. Finally, we provide an overview of our defect localization process.

5.1.1 Target Problem

The proposed technique in this chapter deals with the problem of defect localization. To be more specific, given a target program to be debugged, a failing case, and a set of passing cases, our technique finds methods in the source code that most likely lie on a path to the root cause of the failure. Such methods are good starting points for a root cause search, which, without knowing a proper location to start, can be very complex and costly if not impossible. Our technique also indicates where in the dynamic call tree suspicious methods lie. This suggests subtrees that should be examined to find the root cause.

In this work, we are targeting structure-affecting defects, or defects that cause a structural change in the dynamic call graph. Such defects make the target program diverge from executing the expected path in the source code. Regardless of the reason behind this divergence (e.g., run time exception, hardware failure, or incorrect branch taken as a result of wrong variable assignment or wrong branch condition), as long as the defect changes the structure of the dynamic call graph, the defect can potentially be localized using our approach.

Figure 5.1 illustrates a number of cases in which our approach can localize the defect. In this figure you see sample code, that is the subject of four typical programming defects. The patterns one sees in a correct execution of the code and the ones that may be seen in different failing executions (associated with different defect types) are identified. If the code is correct, any execution of it results in dynamic call graphs (a) or (b). Four defective versions of the code are considered. In case (I), a defect in the code leads to a run time exception in method A. In this case, if the exception is not caught, the resulting call graph observed will be the one tagged

as (I). In other words, none of the methods B, C or D will be executed. In case (II), the branch condition is wrong (\geq instead of $>$). In this case, if $var = 0$ we will see method B instead of method C as in (II). In case (III), a hardware failure happens at the `else` statement. This leads to the execution of both B and C as in (III). In case (IV), the programmer forgot to create a specific branch for the case where $var = 0$. Therefore, instead of executing method E (associated with $var = 0$), method B is executed.

In the current version of the proposed technique, we do not consider frequency-affecting defects, defects that only change the frequency of method calls in a dynamic call graph (e.g., defects that cause a loop to run for too long). Augmenting the tree representation of dynamic call graphs with numbers representing frequency of method calls can be a potential solution to deal with frequency-affecting defects. This idea is investigated as a future work.

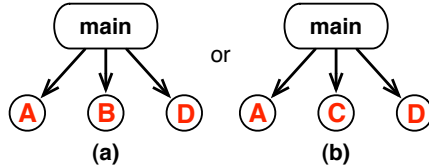
Our approach can localize defects to the granularity of a method. We identify methods which behave suspiciously in terms of the way they call other methods. To get finer grained results, one could replace the notion of “dynamic call graph” in this research with “dynamic control graph”. With this extension, it would be possible to identify a line-of-code (as opposed to a method name) to start searching for the root cause of a failure. This extension is also left to future work.

In this work, we are targeting large systems with multiple features (i.e., functionalities). Most approaches in the literature consider smaller programs and it is not clear if they are able to cope with the challenges associated with large systems, which are usually the target of automatic defect localization. To analyze large systems, we take advantage of feature location, which is a technique used to identify the code

```

main() {
  A (); ← I. Uncaught exception in "A"
  if (var ≥ 0) ← II. Wrong branch condition (≥ instead of >)
    B ();
  else ← III. Hardware failure
    C (); ← IV. Missing branch
  D ();
}
    
```

Correct executions result in:



Faulty executions result in:

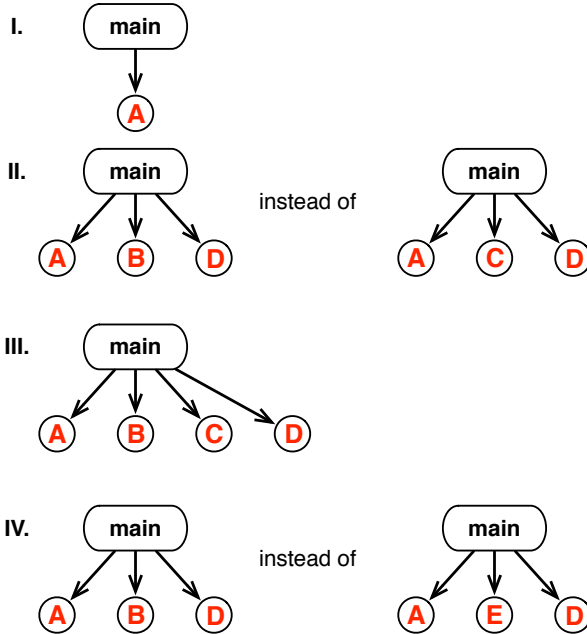


Figure 5.1: Sample code with structure-affecting defects and patterns associated with execution of defect-free and defective code

associated with different functionalities of the system, including the failing feature (which to avoid ambiguity we will call *target feature* in this thesis). This is one of the main distinctions of our work.

In this work, we acknowledge the possible availability of multiple passing test cases but are constrained to work with a single failing test case. Most call-graph-based approaches in the literature consider more than a single failing case. We believe that in a large number of real world cases a software debugger is actually constrained to a single failing case and thus this assumption makes our technique more practical. Clearly this makes defect localization more challenging since we have less information from the target system on which to base our analysis.

5.1.2 Approach

A failing case is a case where the target program crashes, exhibits an unexpected behavior, or provides wrong results. Such a failing case must be documented by two pieces of information:

- Failure setting: is input and scenario which is required to reproduce the failure. For example, a user may report a failure setting by noting that they have filled-in some text box t with some specific value v and clicked on the some button b .
- Failure description: is an explanation of what happens during the failing run; i.e., which parts of the execution are performed as expected and which parts are problematic and also what are the expected results. For example, a user may report that they had expected the system to do tasks x , y and z ; the system

performs task x , encounters a problem during the execution of y and z is never performed.

When one follows a given failure setting and applies it to the target system, the system traverses a path in the code to perform the requested functionalities. This path identifies the methods called by the program (i.e., dynamic call graph) and any defects causing the failure has to lie on this graph. However, a dynamic call graph can be very large. In addition to methods related to the failing functionality, a dynamic call graph may include methods related to system initialization, other functionalities that have been performed successfully, etc. In this work, we represent dynamic call graphs as trees. As will be explained in Section 5.3, this is done via assigning distinct nodes to different calls to the same method. Figure 5.2 illustrates a sample dynamic tree associated with the sequential execution of tasks x , y and z . It indicates how different subtrees can relate to different parts of the execution.

The idea of this work is to first identify which parts of a dynamic tree relate to the failing feature, which reduces the search space for finding a defect. Then, contrast the related subtrees against what we expect them to be, to identify defect-related methods.

A failure description usually provides us with information about what the failing functionalities are. For example if task y is the problematic part of an execution, then we should search for the defect in the subtree associated with y . Now, if we make the execution of a feature frequent within a set of runs, the subtree associated with that also becomes frequent and thus identifiable via frequent-subtree mining. We use a set of passing test cases where the target feature is performed as expected and mine frequent patterns out of their dynamic call trees. Such patterns represent

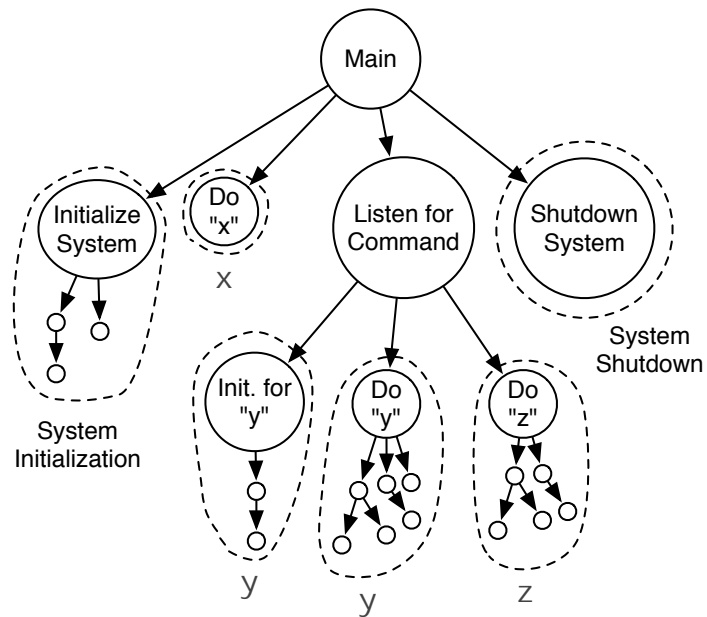


Figure 5.2: Sample dynamic call tree associated with sequential execution of tasks “x”, “y”, and “z”

defect-free execution of the target feature and can be used as a means of identifying what goes wrong in the failing case. We incorporate approximate tree matching to identify whether the failing call tree conforms to the expected patterns. In this context, structural differences are clues that lead us to where the defects lie.

5.1.3 Steps in Proposed Technique

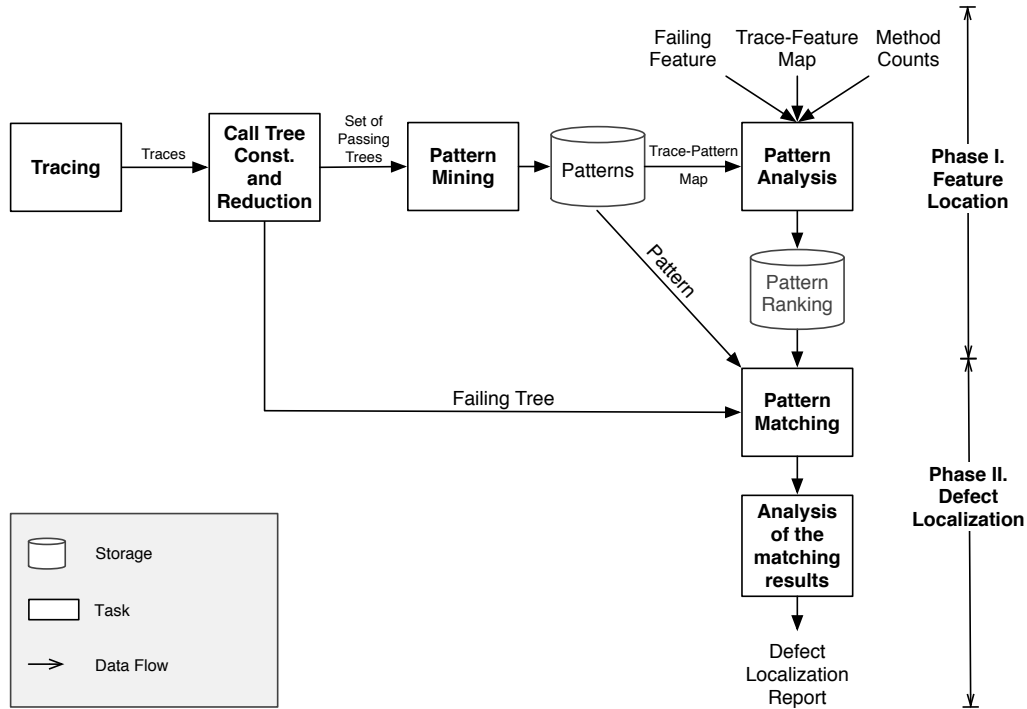


Figure 5.3: Proposed defect localization process

Figure 5.3 illustrates the defect localization process. This process has two major phases. In phase one (*feature location*), we identify patterns associated with defect-free execution of the target feature. This process is further divided into a number of

steps. In the first step, “tracing”, we instrument the target system, run passing and failing test cases, and collect their execution traces (i.e., records of method calls in executions). In the second step, “call tree construction and reduction”, we represent each execution trace as a tree (dynamic call tree), then apply tree reduction techniques to make the trees manageable. In the third step, “pattern mining”, we mine frequent subtrees from call trees of passing executions (also called *passing call trees*). This will result in a database of *passing patterns*. In the fourth step, “pattern analysis”, we identify features exercised by each test case, then given a failing feature, we find and rank patterns associated with that feature. In phase two (*defect localization*), we contrast relevant patterns against the call tree of a failing execution to identify mismatches. The mismatches point to methods in the source code where a search for the root cause of the defect should begin. The following sections describe the process of defect localization in detail.

5.2 Tracing

Tracing is the first step of our defect localization process. It is preliminary work that needs to be done in order to build dynamic call trees, which are the main source of information we use in our analysis. In this section we discuss tracing and its related concepts.

5.2.1 Instrumentation and Tracing

Researchers have used textual, static, and dynamic analyses for defect localization. In this work, we are focusing on dynamic analysis. Like all dynamic analysis-based

approaches, we rely on execution traces to decide which methods may contain defective code. In general, these traces include statements executed, methods called, messages communicated, etc., during an execution.

To record execution traces, we first instrument the target system using an instrumentation tool. Since we use traces for the purpose of building dynamic call trees, in this work we record method entries and exits, the IDs of process and threads executing them, and their associated time stamps, as illustrated in Figure 5.4. Note that, we mainly use this simple trace structure to show the fundamentals of our defect localization in this chapter. However, to further distinguish between methods, in our experiments presented in Chapter 6 we have actually recorded the actual method signatures (i.e., class path, class name, method name, input parameters types, output parameter type) instead of method names.

```
Enter/Exit methodName PID TID TimeStamp  
  
Enter m1 P0 T0 12/11/19-08:10:12  
Enter m2 P0 T0 12/11/19-08:10:13  
Exit m2 P0 T0 12/11/19-08:10:15  
Enter m3 P0 T0 12/11/19-08:10:15  
Enter m4 P0 T0 12/11/19-08:10:16  
Exit m4 P0 T0 12/11/19-08:10:20  
Exit m3 P0 T0 12/11/19-08:10:21  
Exit m1 P0 T0 12/11/19-08:10:25
```

Figure 5.4: Sample execution trace

Available tools for execution tracing differ in the programming language or operating system they can handle, the types of information they can monitor, etc. Depending on the target system, we can use any appropriate tool for dynamic method call tracing. One thing to be aware of is that execution traces can become very large,

depending on the scenario executed on the target system. Large execution traces can take a long time to be produced. More importantly, they can consume run time memory to the point they take up all available memory. Therefore, there is a need for keeping execution traces as small as possible. Filtering is a methodology provided by instrumentation tools to exclude generating traces for certain parts of the execution that are specified by the user. In a filter, one can specify packages, classes, methods, etc., that they wish to include or exclude in the tracing.

At this point, the decision about when and how much to filter is left for the debugger. Factors such as the amount of memory available, the number of test cases used for defect localization, and the size of generated traces affect this decision. Depending on the debugger's understanding of the target system, they can filter out: 1) irrelevant subsystems, packages, classes, or methods, or 2) methods with low probability of being defective, i.e., methods that are small, do not transform data, do not interact with remote components, and are not time dependent (e.g., small utility methods), or methods that have been tested and are proved correct (e.g., library methods). Risks associated with filtering are presented in Appendix A.

5.2.2 Features, Scenarios, and Test Cases

Like many other dynamic analysis based approaches we rely on a set of scenarios to exercise the target system and collect execution traces. In addition to the traces, most defect localization approaches consider the result of an execution, i.e., pass or fail. This information is used along with the execution traces to identify code components distinguishing between passing and failing executions. Besides the execution traces and results, in this approach we record information about the functionalities being

executed in each run and whether they are carried out successfully. This is inspired by the feature location research area, where they typically record functionalities being exercised and components being executed in different runs to identify feature-code relations.

Borrowing from the feature location domain, in this thesis we use the term feature to refer to a functionality provided by the target system and used alone or in combination with other features in a scenario. In contrast to conventional defect localization approaches that do not care about what is actually done in an execution, we exercise a set of meaningful scenarios on the target system and keep track of features executed in those scenarios.

There are two reasons behind this:

1. A major drawback of conventional defect localization approaches is that they do not consider functionalities executed in different runs, and thus potentially aggregate irrelevant information. For example, if a call to method `m` is frequent in failing tests and infrequent in passing ones, they decide that it is very likely that this call is defective. However, this call could simply belong to a functionality that is mostly exercised in failing test cases and has nothing to do with the failure. We argue that considering program features provides richer semantics that can be incorporated to enhance conventional defect localization.
2. Considering features allows us to deal with bigger applications more efficiently. We take advantage of feature location techniques to reduce the code that needs to be analyzed.

Since most features of a system are non-deterministic in terms of the path they traverse in the call tree, our analysis requires the target system to be exercised with

sufficient scenarios to capture different dynamic call trees associated with a feature. In this context, the use of software testing techniques and concepts such as code coverage measures can help to ensure that an adequate slice of the system's set of possible behaviors has been observed. Therefore, to provide a fair amount of coverage we propose using a well-defined suite of test cases to exercise the target system. A test suite has three major benefits:

- Test cases identify the functionalities/steps exercised in an execution. These steps are part of the work that is done in an execution and can be regarded as features in our analysis.
- A proper test suite should provide some sort of coverage on the target system. This provides more control on the amount of code that is visited in our analysis. Completeness of test cases is one of the main concerns of dynamic analysis and this work.
- Test cases are usually provided as a part of the software development process, hence should be available at the time of debugging and we could take advantage of them for defect localization. This will make sure that we are not adding too much burden to what the debuggers already do.

Figure 5.5 illustrates a sample test case used in this work. This test case specifies the process of buying 10,000 securities from a broker. The process starts by a client sending a request to buy 10,000 shares. The broker acknowledges the request and starts filling it in a number of steps. At the end of the day, the broker sends a "Done For Day" message and stops the filling process. This test case identifies the messages

Time	Message Received	Message Sent	Exec Type	Ord Status	Exec Trans Type	Order Qty	Cum Qty	Leaves Qty	Last Shares	Comment
D2.2										
1	New Order (X)					10000				
2		Execution (X)	New	New	New	10000	0	10000	0	
3		Execution (X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution of 2000
4		Execution (X)	Partial Fill	Partially Filled	New	10000	3000	7000	1000	Execution of 1000
5		Execution (X)	Done for Day	Done for Day	New	10000	3000	0	0	Assuming day order. See other examples which cover GT orders

Figure 5.5: Sample test case for a financial system

that are expected to be communicated between different parties of a financial system. In this scenario, the communication of a specific message can be regarded as a feature.

To automate the process of testing, one usually specifies test cases in a certain scripting language. Such a script is called a *test script*. Not only are such test scripts of great help for failure detection, but also we can exploit them for defect localization as they automate the process of tracing. As in feature location approaches, we require that at least one test case in the test suite exercises each feature and each test case exercises at least one feature. Also, no feature is exercised in all the test cases of the test suite (Wilde and Scully, 1995).

5.2.3 A Special Case: Distributed Tracing in SOA-based Systems

Service-Oriented Architecture (SOA) is a set of principles and methodologies for designing and developing software in the form of interoperable services. These services

are well-defined business functionalities that are built as software components that can be reused for different purposes. An SOA-based system consists of a client which connects to and uses services provided in an SOA-based environment. Compared with conventional monolithic systems, collecting trace data from SOA-based systems is challenging (Wilde *et al.*, 2008) for two reasons:

- **Distribution of traces:** in an SOA-based system, execution of a scenario typically involves services that have been deployed on different platforms. Hence, the trace of an execution may be recorded in multiple files. Therefore, for an analysis of the trace data (including building the dynamic call graph) one may require to aggregate multiple trace files. For aggregation we need to know what components are involved in the execution. This may not be known from the start and dynamically be determined at run time.
- **Concurrency of events:** in a service oriented architecture, services are exploited by many concurrent users. Therefore, the trace of a service may interweave data related to different users and hence different scenarios. Figure 5.6 illustrates a sample SOA-based system. There are five services ($S1$ to $S5$) and two clients ($C1$ and $C2$). Now assume the following situation. We run client $C1$. $C1$ calls operation $Op1$ of service $S1$, represented as $S1.Op1$. $S1$ calls operation $Op2$ of service $S2$. On the other hand, there is also some unknown client $C2$ in the environment which invokes $S2.Op2$ through a call to service $S4$. In this case, the trace file generated for service $S2$ contains data related to both clients and it is hard to see which instance of $S2.Op2$ belongs to which client.

As a special case, in this section we are considering the challenges of distributed tracing in SOA-based systems. The mechanism we suggest in this work for aggregating

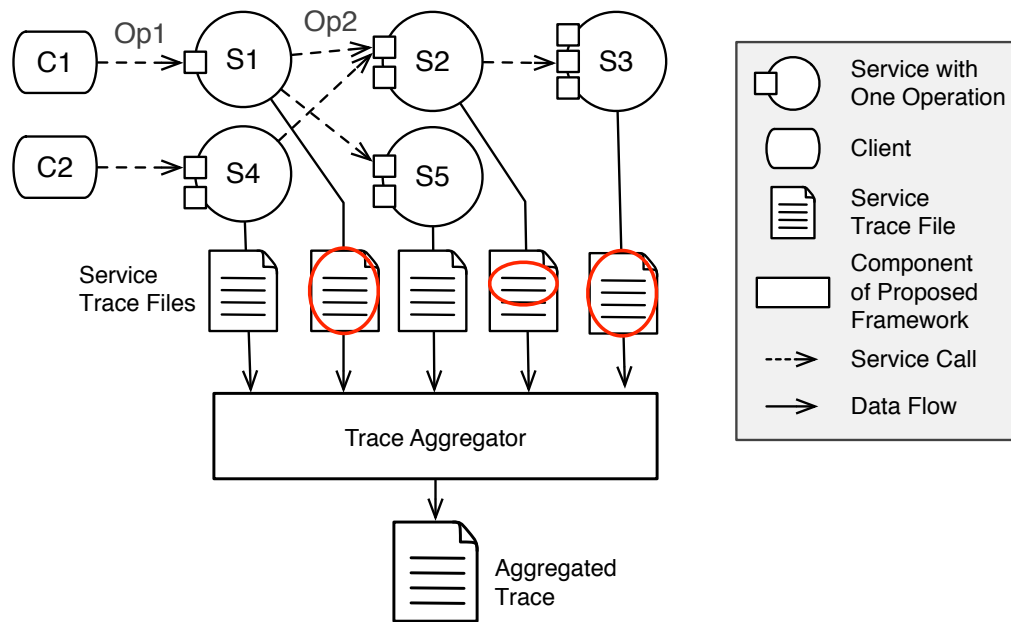


Figure 5.6: Sample SOA-based system

distributed traces exploits three types of causality:

- Time causality: is the before-after relation amongst trace records, based on their timestamps.
- Textual causality: is the use of method names as an indicator of a caller-callee relationship.
- Frequency causality: is the use of the fact that the frequency of observing a certain method call in a trace file is related to the frequency of running a certain scenario.

The aggregation mechanism starts by processing the trace associated with the client program, then we search for instances of service operation calls and download the trace files associated with those services. In this step, we start processing the downloaded traces. In this process, we first use time causality between a caller and a callee, and discard data records timed either before the time of service operation call or after returning from that call. Then, we partition the remaining records into a number of *data blocks*. A data block, is a portion of a trace file which is associated with the execution of a single service operation. We use textual analysis to indicate data blocks. Each data block starts with an “Enter” record, indicating entrance to a method and ends with an “Exit” record, indicating exit from the same method. Using textual analysis, we also detect those data blocks which correspond to an operation different from the one called, and discard them. Next, we assign a unique ID to remaining data blocks and add them to a tree called the block execution tree.

The block execution tree is proposed in this work to aid in the process of distributed trace aggregation. It is defined as a directed rooted tree which represents

the caller-callee relation among data blocks, where the caller is the block which includes a call to a service operation and the callee is the block associated with that service operation. The block execution tree is built gradually as data blocks are detected in trace files. The root of the tree represents the client program and children of the root are the blocks detected in the trace of services that are directly called from the client. Figure 5.7 illustrates an example tree built using this approach. In this figure, (a) shows five trace files associated with services $S1$ to $S5$ as well as the trace file associated with the client program. As you see, the irrelevant trace data is crossed out and the relevant blocks, indicated with ovals, are numbered for unique identification. (b) shows the block execution tree associated with the trace data in (a).

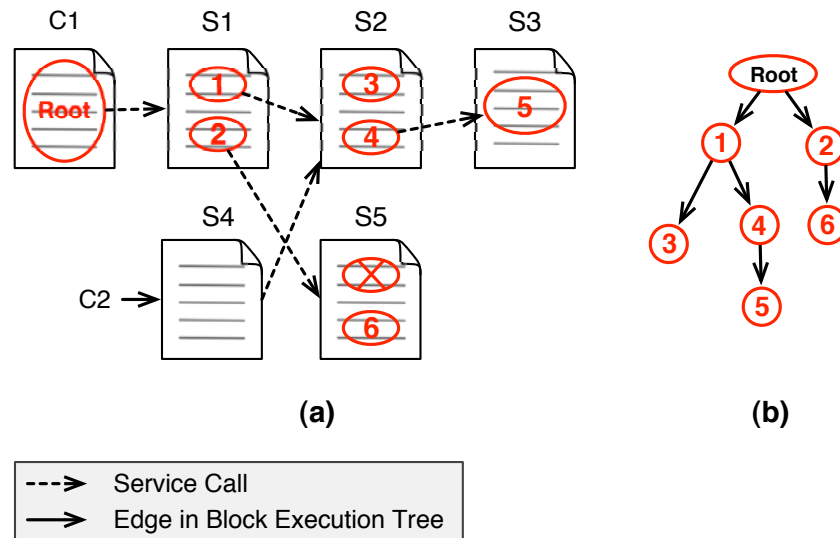


Figure 5.7: (a) Service trace files partitioned into blocks (b) block execution tree associated with (a)

In this stage, we remove the remaining irrelevant data blocks from the block execution tree. As mentioned earlier, a large amount of irrelevant trace data have been discarded using time and textual causalities. However, due to concurrency of events in SOA, there is a possibility that the same service operation is called at about the same time by different users. The blocks generated as a result of such incidences cannot be distinguished via time or textual analysis. We use frequency causality to deal with this issue. In this mechanism, we repeat the execution of the client program with the same input parameters a certain number of times and count the instances of the questionable data blocks. If the number of instances is more than or equal to the number of runs, the block remains, and if not it is deleted from the block execution tree. Respectively, the sub trees starting from deleted blocks are removed. The rationale behind this analysis is that by repeating the execution of the client program, we basically repeat our scenario and respectively each relevant block is repeated in the generated traces. For example in Figure 5.7, blocks three and four belong to different scenarios and it is not clear which one is executed after block one. By applying frequency analysis, we could indicate the relevant block.

Resolving all uncertainties leads to a tree where children of a block are blocks associated with different service calls of the same scenario. In this stage, we merge the traces by replacing each service call with its corresponding block. A depth first pre-order traversal of the block execution tree indicates in which order the blocks should be visited for this reason. In this traversal, nodes are visited in VLR mode (parent-left child-right child).

Details of tracing in SOA systems and the uses of it are provided in the paper (Yousefi and Sartipi, 2011).

5.3 Constructing and Reducing Dynamic Call Trees

As mentioned before, running a scenario on an instrumented system will result in the generation of a trace file which in our case is a record of method entries and exits as illustrated in Figure 5.4. Such a trace can easily be translated into a dynamic call graph. We represent dynamic call graphs as trees, by assigning distinct nodes to different calls to the same method. Compared with the graph representation of dynamic call relations, the tree representation is less compact but more precise in illustrating call relations. For example, as noted by Xie and Notkin (Xie and Notkin, 2002), “a call chain whose length is beyond two is difficult to be extracted from the call graph when one method of that chain, which is neither the head nor the tail of that chain, is called by another method besides the call made by the caller in that chain”.

In this tree representation, we represent each trace as a directed rooted ordered tree (defined in Chapter 3), where nodes represent methods, edges represent method calls, children of a node are methods that are directly invoked from the parent method and are ordered according to their time stamps. The root of the tree is the main entry method (usually associated with the `main` method in a program), and all other methods are children of the root, ordered left to right according to their time stamps.

Although we do not consider concurrency-related defects in the current work, we facilitate the localization of non-concurrent defects in multi-threaded systems. If the target system is multi-threaded, we consider a *multi-threaded call tree*. We represent a *multi-threaded call tree* as a tree with a dummy root whose children are roots of the call trees associated with the threads involved in the execution. Figure 5.8 illustrates a sample trace and its corresponding multi-threaded call tree.

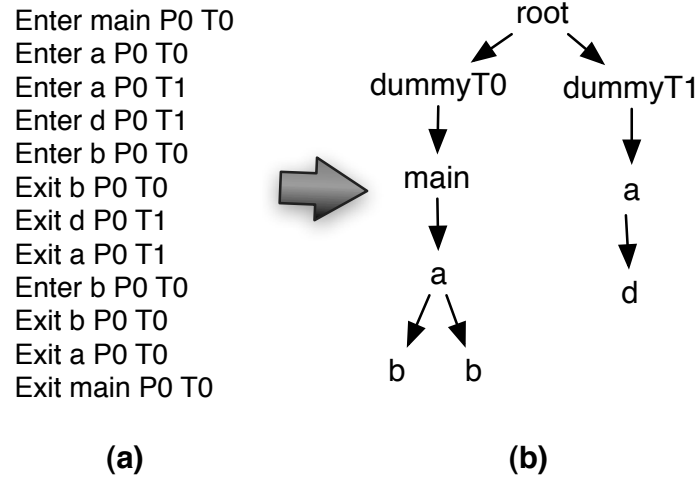


Figure 5.8: (a) Sample trace and (b) corresponding multi-threaded call tree

Dynamic call graphs and specially call tree representations of them can become very big and thus take a lot of memory. To reduce the required memory, it is customary to apply call graph reduction techniques (Eichinger, 2011). In this work we use iteration reduction, where we get rid of replicated method calls resulting from the execution of loops and nested loops. In this case, we keep one instance of a method call in a loop and omit the rest. To perform this, we use a string repetition finder algorithm (Crochemore, 1981) and a cascading mechanism to consequently reduce all repetitive subtrees.

Algorithm 1 presents a pseudo code for cascading iteration reduction. Given a dynamic call tree t , this algorithm produces t' , the iteration-free version of t . In the first step (line 3), the algorithm computes symbolic encodings of all nodes in t . A *Symbolic Encoding* of a tree node in a rooted-ordered tree is a symbolic string which represents children of the node in a unique way. The `setSymbolicEncodings`

Algorithm 1 Cascading iteration reduction

Input: a dynamic call tree t

Output: a dynamic call tree without iterations t'

```

1: procedure CASCADINGITERATIONREDUCTION( $t$ )
2:   repeat
3:     setSymbolicEncodings( $t$ );
4:      $repsFound = \text{False}$ ;
5:      $repsFound \leftarrow$  findAndRemoveRepetitions(root of  $t$ );
6:   until  $\neg repsFound$   $\triangleright$  (no repetitions are found)
7:   return  $t'$ 
8: end procedure

9: procedure FINDANDREMOVEREPETITIONS( $r$ )
10:  findStringRepetitions(symbolic encoding of  $r$ );
11:  if repetitions are found then
12:    remove corresponding subtrees from  $t$ 
13:     $repsFound = \text{True}$ 
14:  end if
15:  for all remaining subtrees do
16:    findAndRemoveRepetitions(subtree's root)
17:  end for
18:  return  $repsFound$ 
19: end procedure

```

procedure, first assigns unique IDs to distinct bottom-up subtrees in t . Then for each tree node in t , it builds a string out of the IDs assigned to the subtrees rooted at the node's children by putting them in the same order as the children themselves. Figure 5.9(a) illustrates a sample tree where unique numbers are assigned to its distinct bottom-up subtrees. It also illustrates symbolic encoding of the tree's root node (i.e., the `main` method), which is string `23437`.

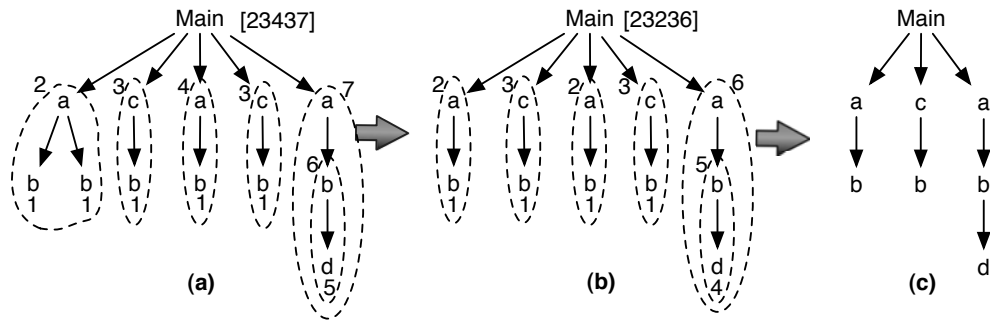


Figure 5.9: (a) Sample tree, unique IDs assigned to its subtrees, and symbolic encoding of its root method (b) the tree after calling `findAndRemoveRepetitions` once (c) the tree after all iterations are removed

In the next step of the algorithm (lines 4 and 5), it searches for iterations in t and removes them. This is done via a call to procedure `findAndRemoveRepetitions`. This procedure incorporates Crochemore's string repetition finder algorithm (Crochemore, 1981) to find repetitions in the symbolic encoding of t 's root (line 10). If repetitions are found (lines 11 to 14), it consequently removes corresponding repetitive subtrees from the tree. Then the algorithm calls 5 again for the remaining children (lines 15 to 17).

To remove less obvious iterations (i.e., nested loops) we call procedure `findAndRemoveRepetitions` several times. A `repeat-until` loop calls this procedure until no

repetitions are found in t any more (lines 2 to 6). Figure 5.9 illustrates a sample call tree before reduction, two passes of `findAndRemoveRepetitions` and the tree after all iterations are removed.

5.4 Mining Frequent Subtrees

Frequent-pattern mining is the essence of our defect localization technique. It is specifically used as part of the process of feature location to identify dynamic call trees representing successful executions of a feature.

A frequent pattern in this work is a subtree that is frequently observed in dynamic call trees of passing traces. It represents a task that is frequently performed in passing scenarios. Therefore, if a feature is exercised frequently (i.e., in greater than a threshold number of scenarios), the resulting patterns include those that represent the frequent feature. Such patterns identify paths in the code that are traversed in successful executions of the feature. Details of this process are discussed in this section. The upcoming section also discusses our pattern analysis approach which explains how one can spot in all possible frequent patterns, those that represent the target feature.

Sartipi and Safyallah (Sartipi and Safyallah, 2010) also used frequent-pattern mining to localize the implementation of software features. Their approach is different from the work in proposed in this thesis: 1) they consider method call sequences as opposed to call trees; 2) they assume that features are deterministic in the sense that they always execute the same sequence of methods.

Eichinger et al., (Eichinger, 2011) also used frequent-pattern mining for defect localization. They use subgraphs obtained from frequent-subgraph mining as different

contexts and perform further analysis for every subgraph context separately. The way we use frequent patterns in this work is different: 1) assumption of single failing cases, means that no frequent patterns can be mined to represent a failure, thus Eichinger's suspiciousness formulae are no longer effective; 2) we reduce the number of patterns to be analyzed by identifying patterns related to the failing feature; 3) we use frequent passing patterns as means of specifying the expected behavior of the failing feature.

The following subsections provide details of our frequent-pattern mining algorithm.

5.4.1 Definitions

In this work a pattern is a *closed frequent bottom-up subtree* of a dynamic call tree. As introduced in Chapter 3, a bottom-up subtree T' of a rooted labeled ordered tree T consists of a node from T (which will be the root of T') and all the descendants of this node (down to the leaves) such that the nodes and edges of T' exhibit the same properties (i.e., labeling and sibling ordering) as in the super tree T .

In frequent-subtree mining, one is given a set of trees TS (also called a *tree-set*) to be mined for frequent subtrees. A frequent subtree is a subtree T' where the cardinality of its enclosing trees from TS (namely, the *frequency* of T') is greater than or equal to a given threshold value, called the minimum frequency. In this case, the subset of trees enclosing T' constitute the *support set* of T' . A frequent closed subtree is a frequent subtree T' , where for any other frequent subtree T'' such that T'' is a super tree of T' , the frequency of T' is greater than the frequency of T'' . We also consider another threshold for subtree mining called *minimum height*, which identifies the minimum height that a subtree must have in order to be considered as

a candidate pattern.

Since we perform pattern mining on the set of passing call trees, the support set of a pattern in this work is the set of passing test cases which exhibit the pattern in their call trees. In the experiments we performed in this study, we consider both minimum frequency and minimum height thresholds to be two because we want to capture as many patterns as possible to reduce the risk of missing any useful ones. The “closed” condition for patterns drastically reduces the number of extracted patterns and thus makes our analysis less costly. It rules out insignificant patterns, those which do not add more to the information provided by the frequent closed subtrees. By definition, any sub tree of a frequent closed tree has the same support as the super tree, and thus not really worthwhile to keep.

5.4.2 Related Work

There is a major difference in the way subtree mining algorithms define a “subtree”. Most of the proposed algorithms are focused on “embedded” and “induced” subtree mining (Chi *et al.*, 2004a). One well-known algorithm for “bottom-up” subtree mining is based on the work by Luccio *et al.*, (Luccio *et al.*, 2001, 2004). It uses the *string encoding* representation of subtrees which is defined in a recursive way as follows: 1) for a rooted ordered tree T with a single vertex v , the pre-order string of T is $PS_T = l_v 0$, where l_v is the label for the single vertex v , and 2) for a rooted ordered tree T with more than one vertex, where the root of tree is labeled r and the children of r are r_1, \dots, r_k from left to right, the pre-order string is $PS_T = l_r PS_T(r_1) \dots PS_T(r_k) 0$, where $PS_T(r_1), \dots, PS_T(r_k)$ are pre-order strings for the bottom-up subtrees $T(r_1), \dots, T(r_k)$ rooted at r_1, \dots, r_k , respectively. The proposed subtree mining algorithm by Luccio

et al., first initializes an array of pointers to each node in a tree-set (note that every pointer points to the root of a bottom-up subtree); then, sorts the pointers by comparing the string encoding of the subtrees to which they point; and finally, scans the array to determine the frequencies of the bottom-up subtrees. Time complexity of this algorithm in all cases is $O(m.n.\log n)$, where m is the number of nodes in the largest tree of the tree-set and n is the number of all nodes in the tree-set. The major drawback of the algorithm proposed by Luccio et al., is that the number of candidate bottom-up subtrees is huge and the cost of frequency counting is high.

In general, most subtree mining algorithms adopt an Apriori-like approach, which is based on the Apriori heuristic for association rule mining (Agrawal and Srikant, 1994), stating that any sub patterns of a frequent pattern must be frequent. The Apriori-like algorithms for subtree mining have two major steps: candidate generation, and frequency counting. The essential idea is to iteratively generate the set of candidate patterns of length $(k+1)$ from the set of frequent patterns of length k (for $k \geq 1$), and check their corresponding occurrence frequencies in the database.

5.4.3 Proposed Algorithm

The proposed subtree mining algorithm in this section is an extension of the bottom-up subtree mining algorithm by Luccio et al. (Luccio *et al.*, 2001, 2004). As mentioned before, the main bottleneck of Luccio's algorithm is that a huge number of candidate subtrees are generated in the first step and the cost of frequency counting is very high. However, a lot of candidate subtrees are infrequent. The main idea behind our algorithm is to find an effective pruning strategy in order to reduce the number of candidates. By considering the minimum frequency property at early stages of

our algorithm, we incorporate the Apriori heuristic to decrease the number of subtrees checked throughout the execution of the algorithm and hence reduce its time complexity. Moreover, the algorithm proposed by Luccio et al., (Luccio *et al.*, 2001, 2004) mines both closed and non-closed bottom-up subtrees. However, we need to distinguish between closed and non-closed patterns. Therefore in the final step, our algorithm indicates and keeps closed patterns and discards the rest.

Algorithm 2 Mining frequent closed bottom-up subtrees

Input: set of trees *treeSet*, minimum frequency threshold *minFreq*, minimum height threshold *minHeight*

Output: frequent closed bottom-up subtrees

- 1: initiate a 2-dimensional linked list whose elements are objects consisting of pointers to root of subtrees in the *treeSet* and their support sets
 - 2: **for** $i = 0$ to $\text{max_height_of_trees}$ **do**
 - 3: tag subtrees with an infrequent child as “infrequent” and delete their corresponding pointers
 - 4: update support sets for level i
 - 5: tag subtrees where $\text{frequency} < \text{minFreq}$ as “infrequent”
 - 6: **end for**
 - 7: remove non-closed patterns
 - 8: **print** pattern info for all patterns starting from level *minHeight*
-

Algorithm 2 presents our subtree mining algorithm. Given a set of trees *treeSet*, a minimum frequency threshold *minFreq* and a minimum height threshold *minHeight*, this algorithm mines frequent closed bottom-up subtrees from the *treeSet*. Steps of the proposed algorithm are as follows.

Step 1 (line 1) - the algorithm initiates a 2-dimensional linked list of pointers that cover all nodes of the *treeSet*, such that pointers in the same row point to nodes in the same *tree level*. The level of a node is defined recursively as follows. Level of a node is equal to maximum level of its children plus one, where leaves are of level zero. Based on this definition, row L_0 contains pointers to leaves, row L_1 contains pointers

to parents with leaf children, and so on. Not that, each entry in the 2-dimensional list points to the root of a bottom-up subtree. In the beginning, the algorithm assumes that all subtrees are “frequent” and marks them accordingly. It also initializes the support set of each subtree with the tree containing its root node. The number of rows in the 2-dimensional list is equal to the maximum tree-height in the *treeSet*.

Step 2 (lines 2 to 6) - the algorithm iterates over the list, starting from row L_0 (i.e., the leaves) and continuing to the top row. It performs the following in each iteration: a) deletes pointers pointing to a node with at least one infrequent child and marks corresponding nodes as “infrequent”; according to the Apriori heuristic sub-patterns of a frequent pattern must be frequent too. Based on this heuristic, deleting infrequent nodes and their ancestors does not change the result of pattern mining. b) Sorts the remaining pointers in a row. This results in identical subtrees becoming adjacent in a row. Then, it scans the row to update the support sets by aggregating the support set of identical subtrees. Note that in our frequent-pattern mining algorithm, we consider two subtrees st_1 and st_2 identical, if there’s a one-to-one relation between their nodes and edges such that, if nodes a and b in st_1 are mapped to nodes a' and b' in st_2 , then edge (a, b) in st_1 is mapped to edge (a', b') in st_2 . Also, the order of children of corresponding nodes in st_1 and st_2 must be the same. c) Scans the row to detect each pointer pointing to an infrequent subtree, by counting its support set members and comparing it with the *minFreq* threshold. Then it marks such nodes as “infrequent”.

Step 3 (line 7) - the algorithm scans the 2-dimensional list from top row to the first (i.e., the leaves in L_0) to delete infrequent subtrees and those whose supports are the same as their parents (i.e., non-closed). For each pointer at a higher row, the

algorithm deletes its children located at lower rows of the list, if the parent and child have same support set. The remaining pointers in the 2-dimensional list indicate closed frequent bottom-up subtrees of the *treeSet*.

Step 4 (line 8) - in the final step, the algorithm prints patterns whose heights are greater than or equal to *minHeight*.

Figure 5.10 illustrates a set of trees to be mined, four iterations of the proposed mining algorithm, and the discovered frequent bottom-up subtrees, where the *minFreq* and *minHeight* thresholds are equal to two. It first shows a 2-dimensional list of pointers to subtrees in the tree set. As the maximum tree height is four, therefore we have four tree-levels and four rows in the 2-dimensional list of pointers. The figure then shows how rows are updated in several iterations in step 2. Discovered frequent closed bottom-up subtrees are also marked on the trees.

Time complexity of the proposed algorithm in the worst case is $O(m.n.logr + n.s)$, where m is the number of nodes in the largest tree in the tree-set, n is the number of all nodes in the tree-set, r is the size of longest row in the list, and s is the number of trees in the tree-set, for the following reasons. Steps 1, 2-a and 2-c each require a single pass over the nodes of the tree-set and thus take a total of $O(n)$ operations; Step 2-b takes a total of $O(m.n.logr)$ operations, where the comparison takes a maximum of $O(m)$ operations, and thus sort takes a maximum of $O(n.m.logr)$ operations; Step 3 is performed in a single pass over the nodes of the tree-set, in which the comparison of support sets takes $O(s)$ operations, thus takes the total of $O(n.s)$.

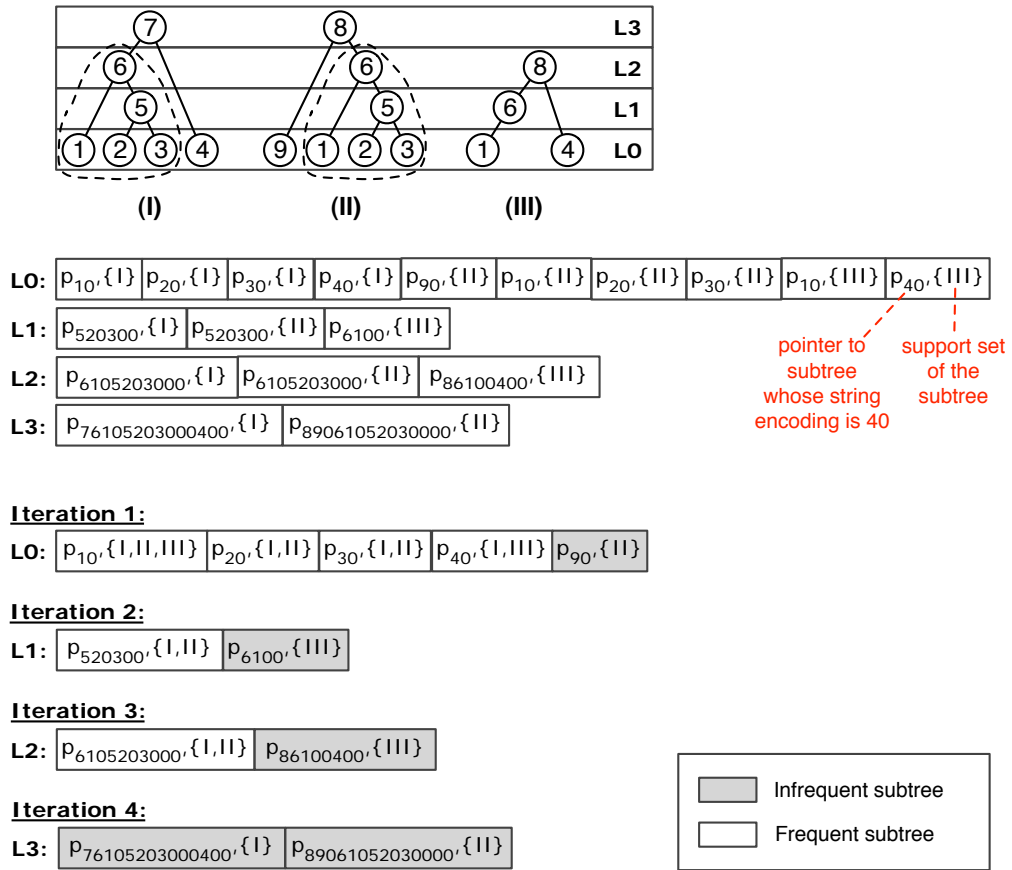


Figure 5.10: Sample tree-set and four iterations of proposed mining algorithm

5.5 Pattern Analysis

Our aim in pattern analysis is to discover the relation between features and patterns. When a feature fails, a developer usually examines the path it has traversed in the code to locate programming defects. Contrasting the call tree of a failing execution with what the developer expects, is a way of identifying suspicious methods. Knowing what to expect in this scenario is not a simple task and requires deep knowledge about functional requirements of the system, algorithms used to satisfy the requirements and the code implementing the algorithms. Pattern mining can help in this process. Frequent passing patterns identify paths that have previously led to successful runs more than a threshold number of times. So, they can be used to identify expectations. However, mining typically produces a lot more patterns than the ones contributing to the execution of the target feature. For example, in Figure 5.11, three patterns are discovered in the pattern mining stage, but only pattern (I) and (II) contribute to the execution of the target (failing) feature, *Acknowledge*. Our ultimate goal in pattern analysis is to distinguish between “feature-specific”, “common”, and “irrelevant” patterns, as explained in the following subsections.

5.5.1 Pattern Semantics

In this section we explain what patterns represent. In other words, we depict the relation between patterns and control structures in the code. A pattern is a bottom-up subtree of a dynamic call tree. It represents the structure of a method in terms of how it calls other methods. Figure 5.12 illustrates different control structures and the corresponding patterns we expect from correct executions of those structures.

A program consists of the following control structures:

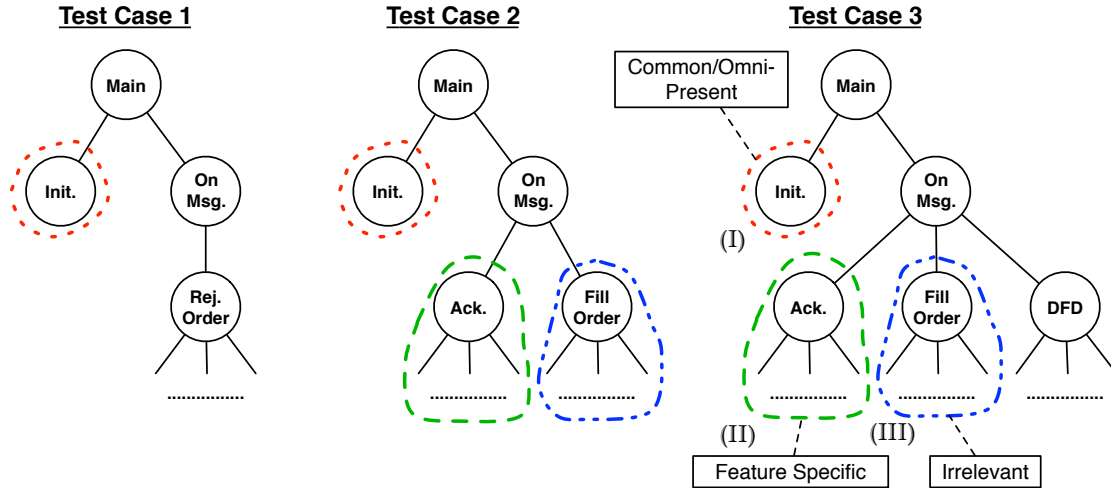


Figure 5.11: Patterns and their relation to target feature, “Acknowledge”

- Sequence: represents statements that are executed sequentially (Figure 5.12-I). In a correct execution of a sequence, we expect a fixed pattern. However, if there are any run-time problems, we may see other patterns.
- Method call: is a control jump from the execution of a caller method to the execution of a callee. Figure 5.12-II illustrates a method call and its corresponding pattern. Recursion is a specific kind of method call.
- Selection (or Alternation): is choosing between alternative paths. Figure 5.12-III illustrates a selection control structure. In this case we have alternative patterns, i.e., exactly one of the alternative methods must be seen in a correct execution.
- Iteration: is repeating the execution of certain statements (Figure 5.12-IV).

Obviously, any complicated structures would be a combination of basic structures,

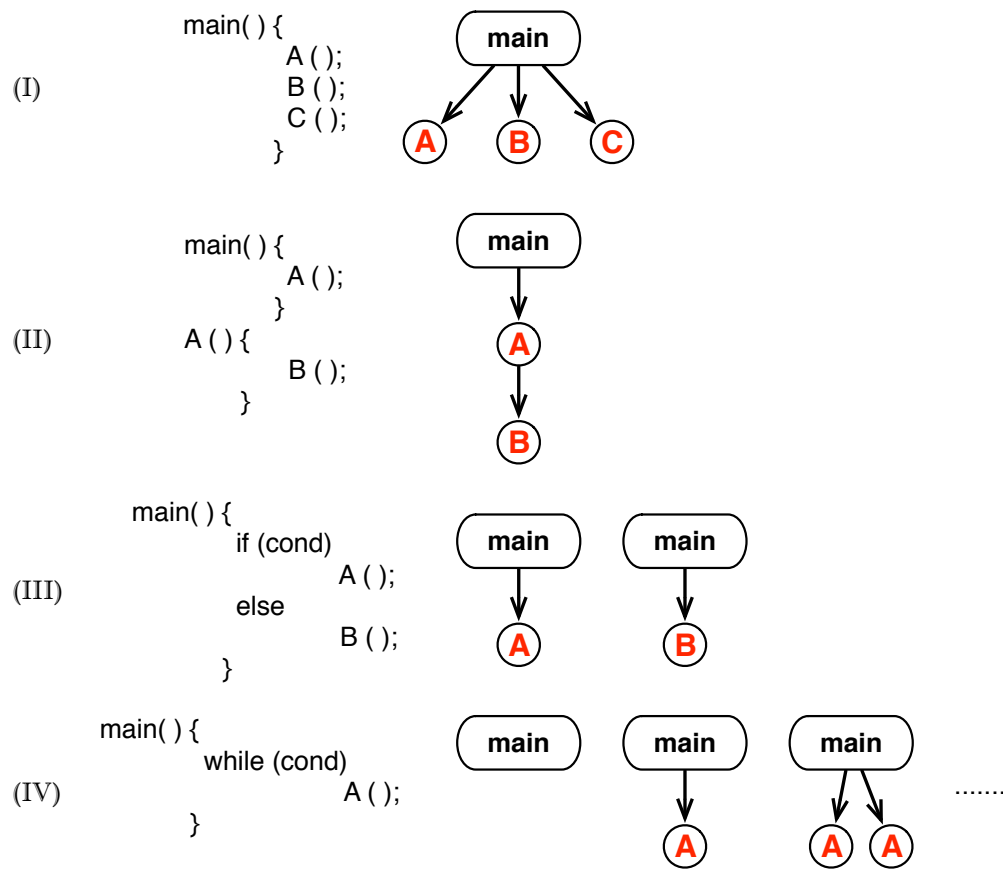


Figure 5.12: Control structures and their corresponding patterns

discussed here, and thus can take advantage of the same discussion.

5.5.2 Pattern Types

As illustrated before, Figure 5.11 presents three types of patterns in relation to a target feature.

- A pattern that is specific to a certain feature f is called *feature-specific*, or *f-specific* for short. It represents core logic of feature f . In other words, it represents a path in the code that is traversed to perform f and is exclusive to f . Pattern (II) in Figure 5.11 represents a feature-specific pattern for a feature called “acknowledge”.
- A *common* pattern represents a shared functionality. In other words, a task that is performed as part of the execution of a number of features. Usually a common pattern represents a low-level subtask such as sorting, math calculations, etc. A specific case of a common pattern is a pattern that is exhibited in all executions. We call such pattern *omni-present*. It usually represents a very basic functionality of the system (such as system initialization, timer related tasks, event handling, and message handling), that is inevitably performed as part of any execution. Pattern (I) in Figure 5.11 illustrates an omni-present pattern.
- An *irrelevant* pattern depicts a non-interesting, random, or unknown shared functionality in a set of runs. For example, if we have both features f_1 and f_2 frequent in our executions, the pattern associated with f_2 is irrelevant to f_1 and vice versa. Also, in some applications such as a picture editor, activities such

as mouse movements could generate meaningless noise patterns. Pattern (III) in Figure 5.11 illustrates an irrelevant pattern.

Based on the above definitions, the following characteristics are assumed for different pattern types.

- A feature f can have more than one feature-specific pattern. f -specific patterns can be related to different subtasks of f or alternatives for the same subtask. Features can have *deterministic* and/or *non-deterministic* subtasks. A deterministic subtask is one which executes the same path in the code in every execution. As a result, it produces the same execution pattern in all runs. A non-deterministic sub-task executes different paths in the code depending on the context and input values. Thus, it produces multiple alternative patterns.
- A pattern cannot be feature-specific for more than one feature or both feature-specific and common or omni-present.

5.5.3 Identifying f -specific Candidate Patterns

As mentioned before, our ultimate goal in pattern analysis is to differentiate between feature-specific, common and irrelevant patterns in relation to the target feature. Deciding about the type of a pattern is essentially a binary decision. However, unless one has a really good insight into the target system's code, it is hard to categorize the patterns. In general, a solution to this problem involves complex analysis of all patterns and features as well as their combinations, which is exponential in nature (See Appendix B). To reduce the complexity of this problem, we change our goal from categorizing the patterns to ranking them in a way that feature-specific patterns get

the highest ranks.

The reason we choose to focus on feature-specific patterns is that we can reformulate common patterns in terms of feature-specific patterns where the feature is no longer “single” but a “set”. In other words, if we know what feature or features fail, by looking at the patterns specific to “the set of failing features” we would be able to localize the defect.

To rank a pattern based on its *specificity* to the target feature, one important clue is the support set of the pattern and the way it intersects with the support set of the target feature (which, similar to the support set of a pattern, is defined as the set of test cases exercising the feature). In this regard:

- A *f*-specific pattern is only exhibited in tests exercising feature *f*. In other words, the support set of *p* is a sub set of the support set of *f*.
- An *omnipresent* pattern is exhibited in all test cases. However, a pattern *common* to a set of features is exhibited only in tests that exercise any feature from the set.
- Any pattern that is only exhibited outside the support set of a feature is *irrelevant*. However, patterns that intersect with a feature’s support set can also be irrelevant.

Based on the above statements, we define a *f-specific candidate pattern* to be a pattern that is only observed in tests exercising feature *f*. In other words, its support set is enclosed in the support set of *f*, or:

$$p \text{ is } f\text{-specific} \iff \text{supportSet}(p) \subseteq \text{supportSet}(f).$$

A sub set of patterns that exclusively includes all f -specific-candidate patterns is called a f -specific candidate set. The term “candidate” suggests that this set includes all f -specific patterns, but potentially can include common and irrelevant patterns too (for example those that are specific to other features, whose support sets intersect with that of f). Further analysis is required to identify the set of f -specific patterns, a sub set of the f -specific candidate set that exclusively includes all f -specific patterns. The same is true if we want to rank f -specific candidate patterns such that f -specific patterns are ranked highest.

5.5.4 Ranking Candidate Patterns

An important clue to help rank the patterns according to their specificity to a target feature is provided by methods inside a pattern. A pattern consists of a set of methods. The idea here is that, if a pattern contains methods that are highly specific to a feature, then the pattern itself must also be specific to that feature. We use and extend *relevance formulations*, from the domain of feature location, to identify to what degree methods are specific to features and consequently the degree of specificity of patterns to features. The following subsections explain different formulations as well as the ranking process.

Related Work

In “feature location”, researchers try to identify in a system, methods that implement a specific functionality (i.e., feature). There are different approaches to this problem. We are interested in statistical dynamic analysis-based approaches, where probabilistic formulations based on dynamic scenario-based information are provided.

Such formulations, called *relevance formulations*, quantify a method’s specificity to a target feature, i.e., to what extent a method is specific to the target feature. This usually facilitates making a fuzzy decision about whether a method is feature-specific, shared among a subset of features, or common to all features.

Software reconnaissance(SR) (Wilde *et al.*, 1992; Wilde and Scully, 1995) is one of the earliest feature location techniques proposed by Wilde and Skully. It analyzes execution traces of two sets of scenarios or test cases: 1) scenarios that activate a target feature, and 2) those that do not. Feature location is then performed by analyzing the two sets of traces using both “deterministic” and “probabilistic” approaches. In the probabilistic approach, they use:

$$\frac{N_C(c_i, f_j)}{N_C(c_i)},$$

where $N_C(c_i, f_j)$ is the number of times component c_i appears in tests exercising feature f_j , and $N_C(c_i)$ is the total number of times that component c_i appears in tests, to compute the relevance of component c_i to feature f_j , and thus identify components that are primarily involved in the implementation of f_j . In the deterministic approach, they use set operations such as set difference or intersection to categorize components based on their relation to the feature. For example, all of the components that appeared in test cases exercising f_j subtracted by those components that appeared in the remaining test cases, would be the ones that primarily implement f_j .

Software Reconnaissance provides a binary judgment based on whether an element is uniquely activated by a feature. As an extension to SR, Edwards *et al.*, (Edwards *et al.*, 2006) propose the *component relevance index*, p_c , which is the proportion of executions of component c_i that occur when the feature is active, that is:

$$P_c = \frac{N_E(fIntervals, c_i)}{N_E(c_i)},$$

where $N_E(fIntervals, c_i)$ is the number of events that are active in $fIntervals$, for all events belonging to component c_i , and $N_E(c_i)$ is the total number of events belonging to component c_i . In this formulation, $fIntervals$ are the time intervals in which feature f is active and an event may correspond to creating an object, entering a method, or executing a code fragment. The use of a numerical relevance index allows one to view the identification of feature-related components as a statistical process, and avoids the all-or-nothing nature of set operations.

Antoniol et al., (Antoniol and Gueheneuc, 2005, 2006; Poshyvanyk *et al.*, 2007) assume a simplified one-to-one mapping between features and components so that events and components can be interchanged. They reformulate Edward et al's relevance index into:

$$P_c(e_i) = \frac{N_E(e_i, f_j)}{N_E(e_i, f_j) + N_E(e_i, \bar{f}_j)},$$

where, $N_E(e_i, f_j)$ is the number of times e_i appears in tests exercising f_j and $N_E(e_i, \bar{f}_j)$ is the number of times the same event appears in the remaining tests. Based, on this formulation, they propose the *Scenario based Probabilistic Ranking (SPR)* where the relevance index has the form of:

$$r(e_i) = \frac{N_E(e_i, f_j)/N_E(f_j)}{N_E(e_i, f_j)/N_E(f_j) + N_E(e_i, \bar{f}_j)/N_E(\bar{f}_j)},$$

where, $N_E(f_j)$ is the total number of events in tests exercising f_j and $N_E(\bar{f}_j)$ is the total number of events in the remaining tests.

As a continuation of SPR, Eaddy et. al. (Eaddy *et al.*, 2008; Eaddy, 2008) proposed *Element Frequency-Inverse Concern Frequency (EF-ICF)* where the relevance metric is defined as:

$$\frac{N_L(l_i, f_j)}{N_L(l_i)} \times \log_2 \frac{N_F}{N_F(l_i)},$$

where $N_L(l_i, f_j)$ is the number of times element l_i appears in tests exercising feature f_j , $N_L(l_i)$ is the total number of times that l_i appears in the tests, N_F is the total number of features, and $N_F(l_i)$ is the number of features that activate l_i .

EF-ICF is based on *Term Frequency Inverse Document Frequency (TF-IDF)* (Baeza-Yates and Ribeiro-Neto, 1999) and *Dynamic Feature Traces (DFT)* scores. TF-IDF is a metric used in information retrieval to indicate relevance of a query to a document based on the terms appearing in the query and the document. EF-ICF essentially borrows the TD-IDF idea to adjust the *DFT* to account for the likelihood that the element is activated by other features.

Our Ranking Formulations

Following the literature on relevance measures we define the following four formulae to compute a method's relevance to a specific feature. We call a quantification of such a relevance the *degree of specificity*. Degree of specificity of method m to feature f , or $DS(m, f)$ is defined as:

- $DS_1(m, f) = \frac{\frac{N_T(m, f)}{N_T(f)}}{1 + \frac{N_T(m, f)}{N_T(f)}}$
- $DS_2(m, f) = \frac{\frac{N_T(m, f)}{N_T(f)}}{\frac{N_T(m, f)}{N_T(f)} + \frac{N_T(m, f)}{N_T(f)}}$
- $DS_3(m, f) = \frac{\frac{N_M(m, f)}{N_M(f)}}{1 + \frac{N_M(m, f)}{N_M(f)}}$
- $DS_4(m, f) = \frac{\frac{N_M(m, f)}{N_M(f)}}{\frac{N_M(m, f)}{N_M(f)} + \frac{N_M(m, f)}{N_M(f)}}$

where,

- $N_T(m, f)$: number of tests that exercise feature f and produce a call to method m in their traces
- $N_T(f)$: number of tests that exercise feature f
- $N_T(m, \bar{f})$: number of tests that do not exercise feature f and produce a call to method m in their traces
- $N_T(\bar{f})$: number of tests that do not exercise feature f
- $N_M(m, f)$: number of calls to method m in tests exercising feature f
- $N_M(f)$: total number of method calls in tests exercising feature f
- $N_M(m, \bar{f})$: number of calls to method m in tests not exercising feature f
- $N_M(\bar{f})$: total number of method calls in tests not exercising feature f

Note that DS_4 is an interpretation of Antoniol et al.'s relevance and the other three formulae are different variations of DS_4 which follow the same concept, but implement slightly different realizations. To compute the degree of specificity values, we record distinct method calls as well as the frequency of each method call in traces and patterns.

Any of these four formulae or any other formula from the literature can be used to identify the method-feature relevance. However, in this work we prefer to use formula DS_2 and DS_1 where DS_2 is used as the primary formulation and DS_1 is used as a tie breaker. There are two reasons for this:

1. We believe that in the quantification of relevance, one should not depend so much on the frequencies of method calls as to whether they are called at all.

Consider the following example. Assume that we have 100 test cases exercising a specific feature f and 100 others that do not. Assume method m appears in all traces exercising f but only in one of the traces that do not exercise f . Also assume that m is called once in each trace exercising f and n times in the trace that does not exercise f (assume that m is called in a loop). In such a setting, we have $DS_2(m, f) = 99\%$. However, depending on n , $DS_4(m, f)$ is varied. For example, if $n = 1$ then $DS_4(m, f) = 99\%$; if $n = 100$ then $DS_4(m, f) = 50\%$; if $n = 200$ then $DS_4(m, f) = 33\%$. As you see DS_4 is not very reliable as it changes as a result of the number of times a loop operates. Also, the above setting is very likely to represent a case where m is shared between features f (exercised in the first 100 tests) and an unknown feature g (exercised in some tests including number 101) such that feature f always requires m but feature g needs it occasionally. We believe that in such a case the likelihood of m belonging to f should be a great number, because it is more relevant to f than to any other features such as g . Therefore, DS_2 seems to be producing more compelling numbers.

2. DS_1 is a good tie breaker when dealing with specificity numbers generated by DS_2 . For example, as long as there are no incidents of a call to m in traces which do not exercise f , DS_2 always produces 1.0, implying that if m is only called in traces exercising f , then it must be specific to f . However, DS_2 can not differentiate between the following cases: 1) m is only called in 1% of traces exercising f ; and 2) m is called in 100% of traces exercising f . Although m may be specific to f in both cases, the latter implies a stronger relation between m and f . DS_1 is a good formula to break the ties in such scenarios.

Pattern Ranking Mechanism

Relevance formulations studied in the previous sections quantify the specificity of a method to a feature. However, what we seek in this work is a way of identifying the specificity of patterns to features. In the following, we explain how we use relevance formulations to rank patterns according to their specificity to a feature.

Patterns consist of methods. To rank patterns according to their specificity to a feature we use pairwise comparison, where pairs of patterns are compared based on the specificity numbers calculated using DS_2 and DS_1 . In this approach, we first descendingly sort methods of each pattern based on DS_2 values where DS_1 is used as a tie breaker. Then, starting from the beginning, we examine the first pair of methods, one from pattern p_1 and the other from pattern p_2 . Comparing these methods using DS_2 where DS_1 is a tie breaker can result in one of the following three outcomes:

- The methods have the same specificity: in this case the algorithm examines the next pair of methods without preferring either of the patterns over the other.
- One method is more specific than the other: the comparison stops. The pattern containing the more specific method is considered more specific. The rationale behind this is that, if a pattern includes a method with high specificity to the target feature then the pattern is more likely to be feature-specific no matter what other methods in the pattern are.
- There's no counterpart method on one of the lists: if the method at hand has specificity value of greater than 0.5, then its corresponding pattern is considered more specific but if it has a specificity value of smaller than 0.5, then its corresponding pattern is considered less specific. In this case, all methods up to this

point have had equal specificities and cannot be used to differentiate between the patterns. The rest of the methods add to the likelihood of specificity, if they are considered specific (greater than 0.5), and deduct from it otherwise (less than 0.5).

When comparing a pair of patterns there is one other thing that needs to be considered and that is the relation between the two patterns in terms of the methods they include. Figure 5.13 uses Venn diagrams to illustrate different cases. In general, we can not decide if one pattern is more specific than the other based on their shared methods. So, if the two patterns we are comparing are of form (b), we ignore their shared methods in the pairwise comparison.

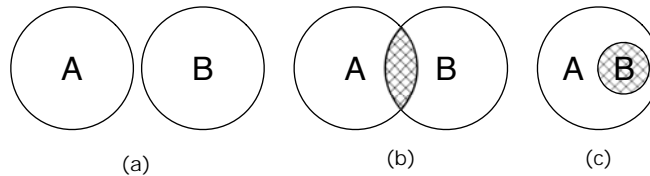


Figure 5.13: Relation of pattern A to pattern B with regard to their methods: (a) patterns have no common methods (b) pattern A and B have some shared methods (c) pattern A includes all methods in pattern B

If the patterns are of form (c), we keep the shared methods for pattern B and ignore them in A , thus comparing the shared part (i.e., B) against the rest of A . This usually represents a case when one pattern is a subtree of the other one. In this case, if the shared part contains methods that are more specific than the rest of the methods, then the smaller pattern is more likely to represent the implementation of the target feature and the bigger to represent a serialization of a number of features besides the target feature. So, the smaller pattern is considered feature-specific. But

if the shared part contains methods that are less specific than the rest of the methods, then the smaller pattern more probably represents a utility function contributing to the target feature and thus the bigger pattern is considered feature-specific. Note that if A and B include exactly the same set of methods, they are equally specific to the target feature as far as this approach is concerned.

Identifying Feature-specific Patterns

As mentioned before, there is no efficient algorithm to decide for certain if a pattern is feature-specific (See Appendix B for details). That, in fact, was the main reason we decided to pursue the pattern ranking approach. Nevertheless, we still need to pick patterns that are very likely to be specific to the target feature for the sake of pattern comparison for defect localization.

A feature in a system is usually implemented using a few core methods. The core methods usually implement the main functionality of the feature. They use other methods to aid them in performing their tasks. Therefore, in terms of dynamic call trees, they are roots of subtrees implementing a feature, i.e., feature-specific patterns. We use this assumption to pick patterns that we believe are more likely to be feature-specific. So, to decide on the specificity of patterns to a feature we consider a threshold number called *number of roots to pick* (*rootsToPick*). In this approach, we pick top ranked patterns with at most *rootsToPick* distinct roots to be feature-specific. In our experiments we considered the *rootsToPick* threshold to be equal to five. Note that this threshold can vary according to the system we are dealing with, for example in a system with only one feature we need to consider all patterns in pattern matching, thus *rootsToPick* is ∞ . Additional experimental study on the use of different values

should be performed as future work.

Other Ranking Criteria

In this section we argue that apart from the specificity measures, other ranking criteria could also be used to identify feature specific patterns:

- Text including method names, comments in the code, etc.: finding keywords related to the failing feature among the method names involved in a pattern or rather contents of the methods (including comments) is a clue that the pattern is likely to be specific to the failing feature. This approach can be very effective depending on how meaningful the names are in the target system's code and how well we can do at guessing the keywords.
- Coverage ratio: in general, the greater the coverage of a pattern over a failing feature, the more likely it is for the pattern to be specific to the failing feature. This works well if features are deterministic. However this is usually not the case. For non-deterministic features, the coverage of a feature-specific pattern tends to be smaller than that of the feature.
- Pattern size: in general patterns that are too small are more likely to perform utility functions and thus be common patterns. This criteria is not always a good one because some utilities are specific to the feature, also patterns that are too big can represent random serialization of features.
- Pattern callers: compared with common patterns, feature-specific patterns usually have fewer callers. However this applies to all feature-specific patterns and

hence distinguishing the ones that are specific to the failing feature needs more analysis.

In this work, we chose a dynamic analysis based approach from the feature location domain, as it is independent from the target system and does not have the above mentioned problems. However, users of our approach can choose or prioritize any other metrics for ranking. In reality, programmers are likely to modify the ranking results based on their understanding of the code and the features.

5.6 Defect Localization

In this section we discuss the process of pattern matching and defect localization. The input to this step is a list of patterns with high likelihood of being feature-specific and the output is a report of the matching results.

5.6.1 Defect Types and their Effect on Dynamic Call Trees

Eichinger et al., (Eichinger *et al.*, 2010b; Eichinger, 2011) identified the following defects as typical programming mistakes that are non-crashing, occasional and dataflow-affecting and/or call-graph-affecting. They concluded this after examining the literature, including the Siemens benchmark programs (Hutchins *et al.*, 1994) that are used in many related publications on dynamic defect localization (e.g., (Jones and Harrold, 2005; Naish *et al.*, 2011; Renieris and Reiss, 2003; Hsu *et al.*, 2008)).

- Wrong variable used. An example uses variable a instead of b in some computation.

- Wrong variable assignment. An example of such a defect is $counter = a + b$ where the correct code is $counter = a$.
- Off-by-one. This defect often happens when accessing the $i + 1^{\text{th}}$ element of a collection instead of the i^{th} or vice versa. For example, $arr[i]$ is accessed instead of $arr[i + 1]$.
- Wrong return value. In this case, only the return statement of a method has a defect. For example, zero is returned instead of *bestValue*.
- Wrong branch condition. This kind of defect covers wrong comparison operators such as $a > b$ instead of $a < b$, additional conditions, missing condition, etc. Furthermore, the Boolean expressions **and**, **or**, **true** and **false**, can be easily misplaced in branch conditions.
- Loop iterations. This kind of defect affects the number of executions of a loop. For example, a for loop uses the wrong counter variable or misses an iteration: `for(int i = 0; i < max; i++)` instead of `for(int i = 0; i <= max; i++)`.

We also add another important defect and that is “missing branch”, where the correct branch does not exist in the code usually because it is not known at design time. This is a very common defect and one of the most difficult to catch.

In this work we focus on defects that change the structure of call graphs. Frequency affecting and data flow affecting defects are not the target of this work. For structure-affecting defects, a defect can lead to:

- Execution of a wrong branch: defects such as wrong variable usage, wrong variable assignment, off-by-one, wrong return value, wrong branch condition,

etc., can make the program take an incorrect branch in the code which itself leads to a program fault.

- Problem in the execution of the correct branch: defects such as wrong variable assignment and off-by-one can lead to run time exceptions which makes the program stop the branch that is being executed.
- Path mixing: things such as hardware defects can lead to generation of unexpected paths in the code. For example, if a hardware failure happens during the execution of an IF statement, the program may execute multiple branches instead of picking one.

As you see structure-affecting defects lead to a deviation from the expected path. The goal of approximate pattern matching is to identify the distance between the path executed by defective code and the one that is expected to execute. This is done through comparing the defective call tree of the failing feature with corresponding feature-specific patterns.

5.6.2 Approximate Pattern Matching

In the approximate pattern matching step we search for previously mined patterns that are specific to the target (failing) feature in the call tree of the failing execution. As we will see in the following section the matching results convey information about the target system and help in localizing the defect.

As our patterns are call trees, in this work we apply approximate subtree matching, which is the technique for finding subtrees that approximately (rather than exactly) match a pattern. In this context, the closeness of a match (i.e., the edit distance)

is measured in terms of primitive operations (i.e., the edit operations) necessary to convert the subtree into an exact match. The edit operations we consider in this work are insertion (indicates that an expected method call is missing from the subject tree) and deletion (indicates that an unexpected method call is observed in the subject tree).

The algorithm we use for approximate subtree matching uses approximate string matching which is similarly defined for strings. This algorithm finds longest common subsequences of two strings and reports their differences.

Algorithm 3 presents our approximate pattern matching algorithm. In this algorithm, we compare a set of patterns (the feature-specific patterns discovered in the previous step) against a subject tree (call tree of the failing execution) to find approximate matches. For each pattern p from the list of feature-specific patterns P , the algorithm:

1. (lines 3 to 6) Searches for the pattern's root method pr in the failing tree t_f . In this stage, it may find zero or more matches. If no matches are found, it reports "root not found" and moves on to the next pattern in the list.
2. (lines 7, 8 and 13 to 20) For each m , pr 's match in the failing tree, which is the root of the bottom-up subtree st_m , the algorithm compares st_m and the tree representation of the pattern p . This is done via a call to procedure `approximateMatching`. In this procedure, we: a) build a string representation of the children of both pr and m . This is done through serializing the symbolic representation of their labels, where each distinct label gets a unique symbol; b) use an approximate tree matching algorithm (i.e., procedure `Diff`) to identify edit differences between the string representations; c) record edit differences and

Algorithm 3 Approximate pattern matching

Input: set of feature-specific patterns P , failing tree t_f

Output: matching results (i.e., list of pattern-match pairs and their associated edit differences, *parentDiffSize* and *diffSize*)

```

1: procedure APPROXIMATEPATTERNMATCHING
2:   for all patterns  $p \in P$  do
3:     search for root of pattern  $p$  in the failing tree  $t_f$ 
4:     if no matches are found then
5:       print root not found!
6:     end if
7:     for all match  $m$  do
8:       approximateMatching( $p$ ,  $st_m$ );
9:       print edit differences, parentDiffSize and diffSize
10:    end for
11:  end for
12: end procedure

13: procedure APPROXIMATEMATCHING( $p$ ,  $t$ )
14:  Diff(string representation of children of  $p$ 's root, string representation of children of  $t$ 's root);
15:  add edit differences to editDiffs
16:  for all matching children  $child1$  and  $child2$  do
17:    approximateMatching( $st_{p,child1}$ ,  $st_{t,child2}$ );
18:  end for
19:  return editDiffs
20: end procedure

```

also parent nodes with different children; d) continue the matching process for matched children through a recursive call to method `approximateMatching` with inputs $st_{p,child1}$, the bottom-up subtree of pattern p with root $child1$, and $st_{t,child2}$, the bottom-up subtree of tree t with root $child2$; e) return the recorded matching results.

3. (line 9) For each p and m , the algorithm computes and prints the total number of mismatches as $diffSize(p, st_m)$. It also, provides the total number of parents with mismatched children as $parentDiffSize(p, st_m)$. Note that $diffSize$ and $parentSize$ for a pattern whose root is not found in the failing tree are equal to one (associated with the missing root).

Figures 5.14 and 5.15 illustrate the matching process by showing excerpts of a sample pattern tree and a subject tree. In the first step of this process (Figure 5.14), we search for the root of the pattern in the subject tree. Then, for each root match we continue the matching as illustrated in Figure 5.15.

5.6.3 Analysis of Matching Results

Approximate pattern matching provides for each feature-specific pattern a list of approximate matches in the failing tree where the edit distance operations (insertions and deletions) are identified for each pattern-match pair. It also assigns two numbers ($ParentDiffSize$ and $diffSize$) to each pair. In this stage we want to look a little closer at what these numbers mean and how we can improve the pattern matching report. Considering the values for $ParentDiffSize$ and $diffSize$, we can identify the following cases for each pattern-match pair:

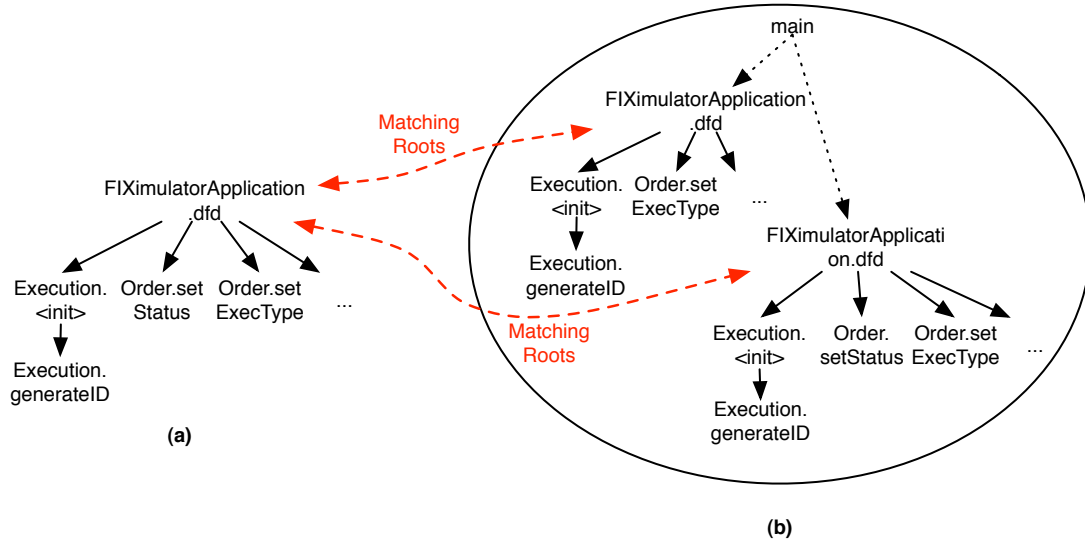


Figure 5.14: (a) A pattern tree and (b) a subject tree

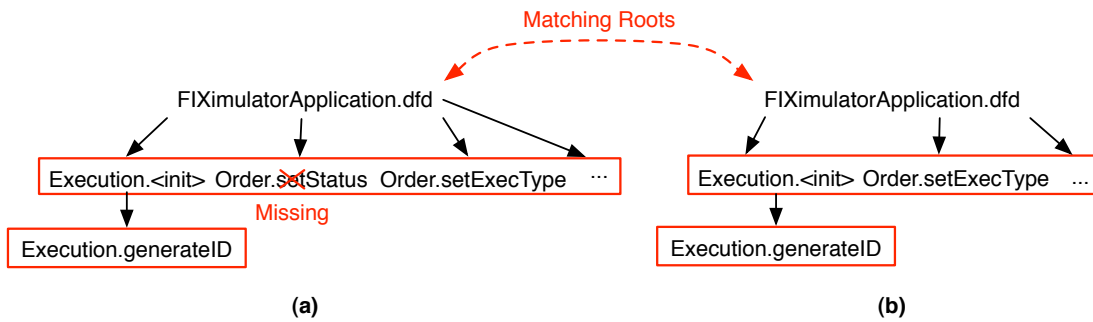


Figure 5.15: (a) pattern tree (b) subtree with matching root from subject tree

- Exact Match ($ParentDiffSize=0$, $diffSize=0$): indicates an exact match. An obvious implication of this may be that the examined subtree (i.e., the match) is defect-free as it conforms to one of the expected patterns. However, this is not quite true. An exact match can also imply that although the observed call tree conforms to one of the expected patterns, it conforms to the wrong pattern due to a defect. In other words, an exact match can happen because of wrong branches being taken in the code thus leading to the observation of a wrong pattern. Referring to Figure 5.1, assume that in a failing execution we observe methods A, B and D. This may indicate that the execution is correct. However, it could also be because of a wrong branch condition as in Figure 5.1-II.
- Root Not Found ($ParentDiffSize=1$, $diffSize=1$): indicates that an expected top method (root of a feature-specific pattern) has not been executed in the failing run. This can point to a defect in the code (e.g., a run time exception, a wrong branch taken). It can also be a false positive, if the execution of the missing method is not necessary for the execution of the feature.
- Approximate Match $ParentDiffSize=m$, $diffSize=n$: indicates that the failing execution does not conform to the expected pattern. This could point to a defect (wrong branch condition, run time exception, etc.). However, it can be a false positive resulting from the comparison of the wrong pattern with the failing tree.

In the approximate pattern matching step, we provided a report of matching feature-specific patterns against the call tree of the failing execution. This report presents an insight into the failing execution and helps the developer to locate the

failure's root. However, the number of feature-specific patterns and the differentiating methods inside them may be large, thus presenting them to the user without a proper ordering is not a good idea.

Many of the defect localization tools provide an ordering of suspicious methods, i.e., a ranked list of methods where those with a higher likelihood of being defective are given higher ranks. To provide such a ranking, the related approaches typically use suspiciousness formulations that quantify the likelihood of defectiveness for a method based on the percentages of its appearance in correct and failing runs. This approach has a couple of drawbacks: 1) Most of the suspiciousness formulations require multiple failing cases. This violates our assumption of having a single failing case. 2) A report involving only a ranked list of suspicious methods provides no context about the potential defects. It only specifies suspicious method names without identifying where and how the methods are called. Identifying methods surrounding the suspicious methods can help in discovering the reason behind the failure.

The report we provided in the previous step, presents a context for suspicious methods as it identifies suspicious pattern-match pairs containing those methods. However, it does not impose a proper ranking on the suspicious pairs. In this stage we sort the pattern-match pairs in this report in a way that pairs that are more likely to reveal a defect are given higher ranks. In this work, we consider two metrics for ranking:

- Coverage of a pattern (i.e., the percentage of tests that exercise the feature and exhibit the pattern in their call trees): the higher the coverage of a feature-specific pattern is, the more it is expected to be seen in an execution involving the feature, hence a mismatch is more suspicious. One good formulation of this

is provided by Dallmeier et al., (Dallmeier *et al.*, 2005), which to the best of our knowledge is the only other work providing a suspiciousness formulation for the case of single failing and multiple passing tests. The formulation he provides is based on sequences of method calls. However, it can easily be reformulated for patterns, as follows:

$$w(p) = \begin{cases} Support(p) & \text{if } p \text{ does not match } CT_{fail} \\ 1 - Support(p) & \text{if } p \text{ matches } CT_{fail} \end{cases},$$

where p is a pattern, $Support(p)$ is the percentage of passing trees exhibiting pattern p , and CT_{fail} is the call tree of the failing run.

- *parentDiffSize* and *diffSize*: the smaller *parentDiffSize* and *diffSize* are, the closer the failing execution is to a feature-specific pattern. This usually implies the existence of fewer defects underlying a single failure. Assuming that every failure corresponds to a few defects or a few defects cause the program to crash and not proceed to pass the other defective parts of the code, we can use this metric to rank the suspiciousness of pattern-match pairs.

Based on the above discussion, we considered the following approaches to sort the pattern-match pairs $[p, m]$ according to their suspiciousness:

- Approach 1. Sort all $[p, m]$ pairs based on *parentDiffSize* where *diffSize* is used as a tie breaker.
- Approach 2. For each match m , ascendingly sort the list of patterns which approximately match m , based on *parentDiffSize* where *diffSize* is used as a tie

breaker. Then, compare head of the associated sorted list of patterns to sort them. Present the results for each m before going to the next.

- Approach 3. Sort $[p, m]$ based on Dallmeier's defectiveness formula. Break the ties using *parentDiffSize* and *diffSize*.

Our initial experiments showed similar results for approaches 1 and 2. Therefore, the results we report in Chapter 6 are mostly based on approach 1. For one of our subject programs, namely *NanoXML*, we also report the results produced using approach 3, which shows improvement over approach 1. However, comparing the approaches requires more investigation, which is one of our future works.

In this step, we provide a ranked report of the matching results to the users. In case we choose approaches 1 or 3, our report would include pattern-match pairs and their differences as illustrated in Figure 5.16. In this figure a pattern is identified by its number and is annotated with its root. If the root of the pattern is not found in the failing tree, the report identifies prospective callers for the root method. Otherwise, a report of missing and additional methods for each approximately matching subtree (identified by its location) is provided.

The report for approach 2 is slightly different:

- if for the first pattern p_i in match m_j 's list, *diffSize* is zero, it means that the pattern p_i observed in the failing execution probably represents an incorrect path followed in the execution, that should not be executed at all or the correct path (if previously seen in correct executions and thus in our pattern database) is one of the other patterns in m_j 's list. So, in our report for each pattern p_i we report that method m_j (which is also root of pattern p_i) is problematic and it

either should not be called at all or the path it executes is incorrect (in which case we provide the edit operations we computed in the previous step for other patterns in m_j 's list).

- if for the first pattern p_i , $diffSize$ is greater than zero, the above argument still holds. The only difference is that we also present the edit operations for p_i in our report.



```

*** pattern p213 (root= edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.dfd(Ledu/harvard/fas/zfeledy/fiximulator/core/Order;)V)

*RNF: root of the pattern not found.

Expecting: edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.dfd(Ledu/harvard/fas/zfeledy/fiximulator/core/Order;)V

Prospective callers: edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.onMessage(Lquickfix/fix42/NewOrderSingle;Lquickfix/SessionID;)V

*** pattern p203 (root= edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.sendExecution(Ledu/harvard/fas/zfeledy/fiximulator/core/Execution;)V)

*Location 4701 - missing calls: 10, additional calls: 0, diffSize: 10, parentDiffSize: 1

missing< edu/harvard/fas/zfeledy/fiximulator/core/Order.getQuantity()D

from edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.sendExecution(Ledu/harvard/fas/zfeledy/fiximulator/core/Execution;)V

missing< edu/harvard/fas/zfeledy/fiximulator/core/Execution.getDayOrderQty()D

from edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.sendExecution(Ledu/harvard/fas/zfeledy/fiximulator/core/Execution;)V

missing< quickfix/field/DayOrderQty.<init>()D)V

from edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.sendExecution(Ledu/harvard/fas/zfeledy/fiximulator/core/Execution;)V

missing< quickfix/fix42/ExecutionReport.set(Lquickfix/field/DayOrderQty;)V

from edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.sendExecution(Ledu/harvard/fas/zfeledy/

```

Figure 5.16: Defect localization report

Compared with the ranked list of suspicious methods, this kind of report provides more information to a user. It provides not only a location to search for the root cause, but also expected and observed paths, which guide the user to where the defects may locate. However, to compare our results with related work an extra step is needed, which is to provide a ranked list of suspicious methods. We do this by looking at the ranked list of pattern-match pairs and taking note of the methods involved, as follows:

- in case of a root-not-found, we note the missing root's prospective callers.
- in case of a match, we note the root of the matching pattern.
- in case of a mismatch, we note the parents with different children.

The list constructed in this way will identify starting locations to search for the root cause of the failure.

5.7 Discussion

In this chapter, we proposed a novel defect localization technique. The proposed technique deals with structure-affecting defects, i.e., defects that cause the execution of an unexpected dynamic call tree. In this technique, we assumed the availability of one failing test case and multiple passing test cases. We first mined dynamic call trees of passing test cases. The frequent patterns discovered in mining are subtrees representing defect-free execution of system functionalities. We then analyzed these frequent patterns to discover their relations to different functionalities of the system. Having a failing test case we built a dynamic call tree of the failing execution and

searched for subtrees that approximately matched a select number of relevant patterns. A relevant pattern in this case is a pattern that is associated with the correct execution of the target (failing) feature. A report of the matching results was provided to the user.

The strengths and shortcomings of the proposed technique are presented in the following subsections.

5.7.1 Benefits

The current work provides a number of advantages over related approaches:

- No database of failing test cases is needed. The assumption of a single failing case in this work makes defect localization more challenging. In the real world, there are many programming defects that cannot be linked to multiple test cases. In many scenarios there exists only a single test case that reveals a specific defect in the code. Also, sometimes a defect is not caught in the testing phase and is later reported by end users of the system. Such occasional defects reveal themselves in very specific circumstances and thus are more challenging to localize. In this sense, our approach is more powerful and practical than spectra-based approaches which require a database of failing executions.
- We deal with multi-feature systems. The proposed technique in this paper is tailored for multi-feature systems and takes advantage of feature location to reduce the code that needs to be analyzed.
- We provide context for debugging. In addition to names of the defect-related methods, our approach identifies defect-related patterns and provides a report

of probable reasons behind the failure (in terms of what method calls are or are not expected). This provides a context for developers to better understand the failing feature as well as the defect and makes fixing defects easier.

- We manage non-determinism. In contrast to other call graph/tree based approaches we do not depend on the existence of fixed patterns in passing and failing executions and deal with non-determinism in feature execution.

5.7.2 Risk Analysis

Appendix A presents a collection of risks associated with our defect localization technique and possible mitigations of those risks. Risk analysis is an important step that needs to be done in any project to identify potential pitfalls and plan for them. We believe that new techniques or tools provided in the software industry should also be accompanied with proper risk study to indicate what could go wrong, thus preventing the new technique or tool to successfully complete its tasks as desired.

In the analysis we have performed in this work, we have identified risks associated with every step of the proposed defect localization technique. We also have indicated the consequences in case those risks happen. We then suggested possible mitigations for the identified risks. Most of the identified risks stem from the dynamic nature of the analysis we use for defect localization.

Dynamic analysis usually deals with large amount of information produced at run time. Risks such as the possibility of long tracing times and the generation of large traces and their corresponding call graphs/trees, relate to this aspect of dynamic analysis.

Dynamic analysis also suffers from incompleteness, as the assumption of having

test cases that cover each and every path in the source code of a system is not logical. Consequently, we can not count on the completeness of the discovered patterns. This may lead to inability of the proposed technique to locate useful defect-related methods in some scenarios. However, in most cases the provided defect localization report is to some extent useful for identifying the defects.

Techniques such as filtering specific methods in the instrumentation phase or reducing iterations in call graphs/trees, which are utilized in dynamic analysis-based approaches to reduce the amount of information that needs to be analyzed may lead to the loss of useful information that is required to locate suspicious methods in a failing execution. We must be cautious when applying such reduction techniques. We recommend the investigation and development of other less troublesome techniques to handle large traces.

There are a number of risks associated with improper ranking of patterns (to identify feature-specific patterns) and pattern-match pairs (to identify suspicious matches/method calls). These risks can lead to situations in which defect-related methods are ranked low down the final list of suspicious method calls, so that it is practically not useful.

Furthermore, there exists risks related to one of the underlying assumptions of this work, that the defects to be found make the call tree of the failing execution different from its correct version.

Note that there are no one-size-fits-all solutions for defect localization in the literature. Every approach can deal with specific defect types under certain assumptions. To build a powerful defect localization tool, one should combine different approaches.

Chapter 6

Evaluation

In this chapter we present an experimental evaluation of our defect localization technique. To show the applicability of and to evaluate our technique, we implemented a prototype defect localization tool. Then we conducted a set of experiments, in which we used our tool to localize numerous defects on four subject programs. In this chapter, we present the experimental results. For our first subject, *FIXImulator*, we also provide details of the defect localization process. But, for the other three subjects we only present the final results, then compare them to related approaches.

The rest of this chapter is organized as follows. Section 6.1 provides details about the prototype implementation of the proposed technique. In Section 6.2 we introduce the programs we use as subjects of defect localization. Section 6.3 introduces the evaluation measures we use in the different experiments to present the results and compare them with the related approaches. Section 6.4 introduces the instrumentation tools that we use in this study to collect the required execution traces. Section 6.5 provides snapshots of how our tool works with the first subject, *FIXImulator*. Section 6.6 provides results and comparisons related to the second subject, *Weka*.

Section 6.7 provides results and comparisons related to the third subject, *NanoXML*. Section 6.8 provides results and comparisons related to the fourth subject, *print_tokens*. Section 6.9 provides the concluding remarks.

6.1 Prototype Implementation

As a proof of concept, we built a prototype tool for defect localization which uses the technique proposed in Chapter 5. Our defect localization tool is implemented in Java and deployed on a 64-bit virtual Ubuntu (version 10.04) machine with 6.8 GB of RAM and a 2.93 GHz Intel Core i7 CPU. Figure 6.1 illustrates the architecture of our defect localization tool. The following describes different components of this architecture.

- *Trace Manager* is responsible for parsing execution traces, preparing them for *Tree Manager*, which constructs dynamic call trees from the traces. *Trace Manager* also provides a report of the frequencies of method calls in traces, for *Feature Location Engine*.
- *Tree Manager* builds dynamic call trees from traces. It also performs iteration reduction.
- *Pattern Mining Engine* mines frequent bottom-up subtrees from the set of passing trees. It stores the patterns as well as the support sets of patterns in *Patterns Database*, which in the current version is implemented as both XML and plain text.
- *Feature Location Engine* ranks patterns in the *Patterns Database* according to their relevance to a target feature. It uses the method call frequencies report,

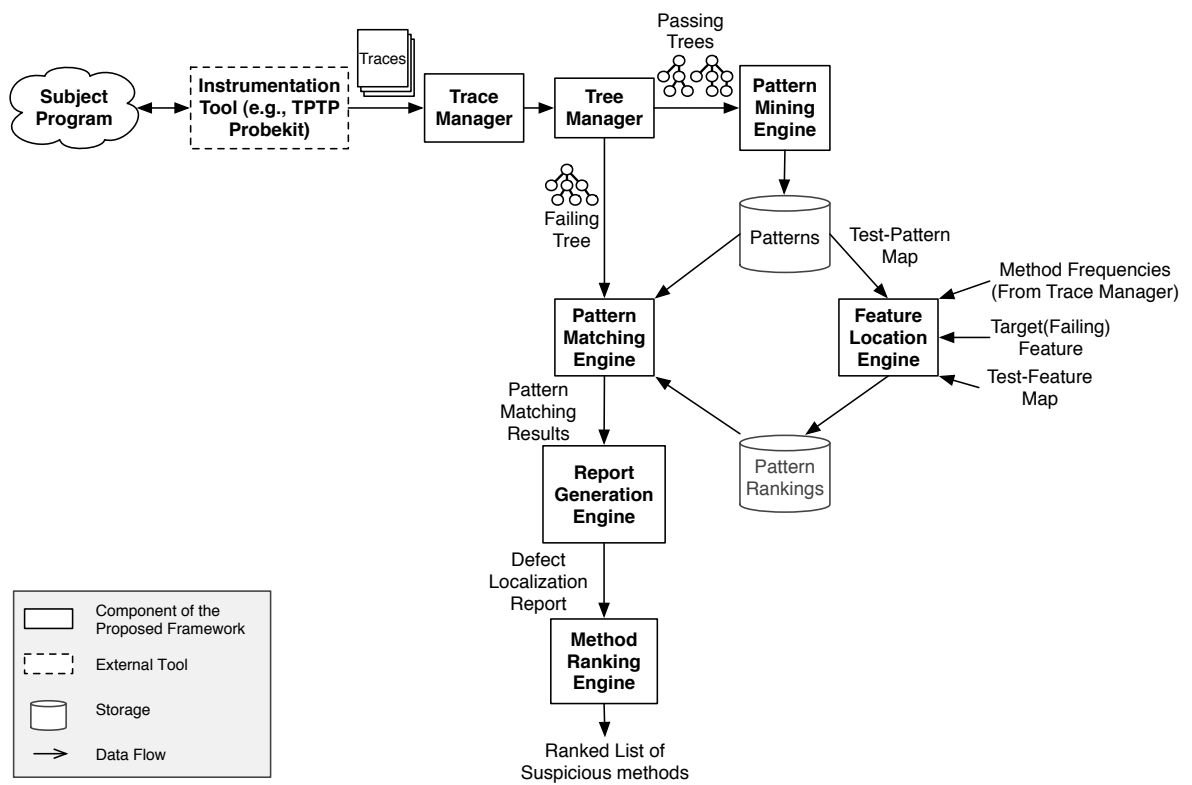


Figure 6.1: Architecture of proposed defect localization tool

test-pattern map (a mapping of patterns to tests which exhibit the patterns in their associated call trees; this is built using the support set information for the patterns, which is stored in the *Patterns Database*), and test-feature map (a mapping of features to tests exercising them; this should be provided as an input to the tool) to produce a ranked list of candidate feature-specific patterns, using the ranking technique introduced in Section 5.5. It then uses the *rootsToPick* threshold to identify patterns that are very likely to be feature-specific.

- *Pattern Matching Engine* approximately matches a feature-specific pattern against the call tree of the failing execution and provides a report of the matching results, including the edit operations.
- *Report Generation Engine* uses the matching results associated with the feature-specific patterns to generate a defect localization report. This report includes a sorted list of pattern-match pairs, ranked according to their likelihood of defectiveness.
- *Method Ranking Engine* uses the previous defect localization report to provide a sorted list of methods, ranked according to their likelihood of defectiveness.

6.2 Subject Programs

To show the applicability of, and to evaluate the proposed technique, we conducted a set of experiments. We targeted a number of subject programs with a controlled set of defects to examine how our defect localization tool works. We chose programs of different nature, in terms of the functionalities they provide, their sizes, the languages

used for their implementation, the types of defects they contain, to see whether we are able to localize defects in different settings.

Although defect localization literature is very extensive, there is no agreement among researchers on the subject programs they use for their experiments. In fact, except for the *Siemens suite* and a few other small programs such as *Space* (Jones *et al.*, 2002), there are no widely used benchmarks in this domain. Benchmarks such as the *Siemens suite* are too small to represent real programs. Therefore, many approaches, especially those targeting bigger programs, use a wide variety of other subject programs to illustrate their capabilities. These subject programs include *Tornado* (Tornado, n.d.), *WebLech* (WebLech, n.d.), *Mozilla Rhino* (Rhino, n.d.), *Weka* (Weka, n.d.), *NanoXML* (NanoXML, n.d.), as well as other programs developed by the researchers themselves.

The technique proposed in this thesis is also more effective when applied to programs with multiple functionalities. Thus, we also needed a larger subject program with multiple functionalities to experiment on. To be able to perform comparisons with the related work, as well as illustrate the capabilities of our technique, we incorporated four subject programs in our experimental evaluation: *FIXImulator* (FIX-Imulator, n.d.), which is a multi-feature program and has been seeded with a number of defects by our collaborator, the Systemware Innovation Corporation company (SWI); *Weka* (Weka, n.d.), which has also been used by Eichinger *et al.*, (Eichinger *et al.*, 2010b); *NanoXML* (NanoXML, n.d.), which has also been used by Dallmeier *et al.*, (Dallmeier *et al.*, 2005); and *print_tokens*, one of the seven programs from the *Siemens suite*, which has been used by many researchers in this area (Jones and Harold, 2005; Yu *et al.*, 2011; Naish *et al.*, 2011; Renieris and Reiss, 2003; Hsu *et al.*,

2008). Our subject programs are introduced in the following subsections.

6.2.1 *FIXImulator*

FIXImulator (FIXImulator, n.d.) is an open source Java-based program, which is part of a distributed capital markets trading system. It is a sell-side trading program which uses the *Financial Information eXchange (FIX)* protocol to exchange securities transactions. FIX protocol is a messaging standard developed for the real-time exchange of securities transactions. It is widely used by both the buy side (institutions) as well as the sell side (brokers/dealers) of the financial markets (e.g., investment banks, brokers, stock exchanges, etc.) for automated trading. As illustrated

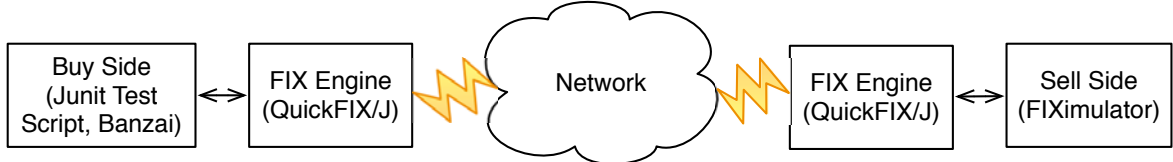


Figure 6.2: Architecture of target FIX-based trading program

in Figure 6.2, a sample FIX-based trading system is composed of three components communicating via FIX messages:

- *FIX Engine* is a messaging engine, managing FIX messages. It performs message validation, sending, receiving, etc. QuickFIX/J is a full-featured open-source implementation of the FIX engine in Java. It supports different versions of the FIX protocol (e.g., FIX 4.0-4.4).
- *Buy Side* is the initiator of securities transactions. A transaction starts when a

buy-side program sends a request to buy securities. The QuickFIX/J package provides a sample buy-side program called *Banzai*. It also provides a set of JUnit acceptance test cases to test the trading system. In this study, we use both *Banzai* and the acceptance test suite to initiate transactions.

- *Sell Side* is a party responding to buy queries by rejecting or filling them. *FIXImulator* is a Java based sell-side FIX trading program whose main purpose is to be used as a testing tool in writing, building, and analyzing buy-side trading programs that use FIX. In this study, *FIXImulator* is used to communicate FIX messages with *Banzai* or the acceptance test cases.

In this study we focused on *FIXImulator*. In other words, we investigated defects in *FIXImulator*'s code and only instrumented this component for defect localization. Defects were introduced to *FIXImulator*'s code by our collaborator, the SWI company. They provided 29 defective versions of the code, each containing one (versions 1-10, 13-24, 26-29) or more defects (versions 11, 12, and 25). Note that a single defect does not necessarily correspond to a single change in the code. In many of the defective versions, more than a single line of code is changed, but not all changes lead to a failure. Moreover, in versions 22-25 the changes have been applied to different methods. The embedded defects cover a wide range of programming problems such as, 5x wrong variable assignments, 2x wrong branch conditions, 3x missing or extra branches, 4x wrong method calls, 13x missing or additional method calls, 4x wrong method parameter values, 1x wrong ordering of method calls.

FIXImulator is a medium-size program which provides multiple functionalities and thus is a good subject to show the capabilities of our defect localization tool. To the best of our knowledge, no other approaches have targeted this program before.

So, we will not provide any comparison results for *FIXImulator* and use it as a case study to examine different steps of our defect localization technique. Table 6.2.1 provides statistics about *FIXImulator*.

Table 6.1: Code counts for FIXImulator

Subject Program	No. Classes	No. Methods	LOC	No. Defective Versions
FIXimulator	29	669	8554	29

6.2.2 *Weka*

Weka (Weka, n.d.) is an open source data mining software, developed at University of Waikato. It is a collection of Java-based machine learning tools for data mining tasks such as pre-processing, classification, regression, clustering, association, and visualization.

Eichinger et al., (Eichinger *et al.*, 2010b) used *Weka* to evaluate their defect localization tool. They manually inserted a number of defects into *Weka*'s source code, thus providing 16 defective versions of the code. The defect types introduced in *Weka* are typical programming mistakes, are non-crashing, occasional and dataflow-affecting and/or call-graph-affecting. In total, ten separate defects (Versions 1-10) as well as six combinations of two of these defects (Versions 11-16) have been introduced in the code. The defects introduced are 4x wrong variable assignments, 3x wrong return values, 1x off-by-one, 1x wrong loop condition and 1x wrong branch condition.

Weka is a multi-feature program providing different data mining related functionalities. It is a large program consisting of more than 19k methods. The complexities of this program makes it an interesting case for study. Also, we use this subject to

compare our tool with the tool provided by Eichinger et al., (Eichinger *et al.*, 2010b).

Table 6.2.2 provides some statistics about *Weka*.

Table 6.2: Code counts for Weka

Subject Program	No. Classes	No. Methods	LOC	No. Defective Versions
Weka	1355	19395	291407	16

6.2.3 *NanoXML*

NanoXML (NanoXML, n.d.) is a small XML parser implemented in Java. Five defective versions have been generated for *NanoXML*, each containing multiple defects. *NanoXML* has a total of 33 known defects. The defects were discovered during the development process, or seeded by Do et al (Do *et al.*, 2004) and others. A defect definition file identifies defects in each version. Unfortunately, no defect definition files are provided for Version four of *NanoXML* and thus we are not able to provide results for this version.

NanoXML is a small program. The only functionality provided by *NanoXML* is parsing an XML file. Therefore, *NanoXML* is not really the target of our defect localization technique. However, as Dallmeier et al., (Dallmeier *et al.*, 2005) also deal with a similar problem as targeted in this thesis (i.e., defect localization using only one failing execution), we also use *NanoXML* as a subject program to be able to compare our results with what they have published. Table 6.2.3 provides statistics about *NanoXML*.

Table 6.3: Code counts for NanoXML

Subject Program	No. Classes	No. Methods	LOC	No. Defective Versions
NanoXML	21-26	177-286	2313-3696	5

6.2.4 *Print_tokens*

Print_tokens, one of the seven programs in the *Siemens suite* benchmark, is a lexical analyzer which tokenizes its input file and prints out the tokens. The *Siemens suite* benchmark consists of a set of programs, which were assembled by Tom Ostrand and colleagues at Siemens Corporate Research for a study of the defect detection capabilities of control-flow and data-flow coverage criteria (Hutchins *et al.*, 1994). *Print_tokens* comes in seven defective versions and each of these versions has one or more defects injected. The goal of defect seeding is to introduce defects that were as realistic as possible, based on experience with real programs. For each base program, the researchers at Siemens created a large test pool containing possible test cases for the program. The researchers retained only defects that were neither too easy nor too hard to detect, which they defined as being detectable by at most 350 and at least three test cases in the test pool associated with each program.

Print_tokens is a small program written in C. It consists of only one feature. Thus, it is not the target of our defect localization technique. However, as the Siemens benchmark (including the *print_token* program) is a reference of many researchers in this domain, we would also like to perform some experiments on it. Table 6.2.4 provides statistics about the *Print_tokens* program.

Table 6.4: Code counts for print_tokens

Subject Program	No. Classes	No. Methods	LOC	No. Defective Versions
print_tokens	NA	18	344	7

6.3 Evaluation Measures

As discussed in Chapter 5, similar to most defect localization approaches, our technique can also produce a list of methods, sorted according to their likelihood of being related to the failure. In this section we introduce the measures we use in this experimental study to evaluate the resulting list. Various measures have been suggested in the literature to assess the precision of defect localization techniques. We use the following in this work.

- *Position*: position of the defective method in the ranked list. This measure quantifies the number of methods a developer has to review in order to find the defect in a random case. Assuming that all methods have the same size and the same effort is required to localize a defect within a method, this measure represents the effort one must invest to find the defect. Smaller numbers are preferred. The assumptions may not be realistic, but *position* still seems a useful measure in comparing the results obtained by competing methods.
- *Rank*: position of the defective method in the ranked list, where methods with the same likelihood numbers use the worst position for all methods with the same likelihood. This measure quantifies the number of methods a developer has to review in order to find the defect in the worst case. Assuming that the effort required to examine different methods is about the same, this measure

represents the effort one must invest to find the defect in the worst case. Smaller numbers are preferred.

- *Number of methods atop*: *Rank* of the defective method minus one. This measure quantifies the number of methods a developer has to review before reaching the actual defective method. Similar to the *Position* and *Rank* measures, this measure also represents the effort one must invest to find the defect. Smaller numbers are preferred.
- *Score(M)*: percentage of methods that need not be examined (i.e., those that appear after the defective method in the list). *Score(M)* is computed as:

$$Score(M) = \frac{Total\ Number\ of\ Methods - Rank\ of\ the\ Defective\ Method}{Total\ Number\ of\ Methods} * 100\%.$$

Greater numbers are preferred.

- *Distance to root cause*: in the case the reported method is not the actual defective method, we provide the *distance to root cause* measure, which identifies the number of edges in the dynamic call tree which sets the root cause apart from the reported method. For example, if the reported method is parent or child of the actual defective method, the *distance to root cause* is equal to one. Smaller numbers are preferred.

In the above calculations, if more than one defects exist in the code, we report the numbers associated with the best ranked defect. This reflects that a developer would first fix one defect, before applying our technique again. This is consistent with other defect localization approaches in the literature.

6.4 Instrumentation

As mentioned before, we perform program instrumentation to collect execution traces, which are the main inputs to our defect localization tool. Different instrumentation tools have been developed, that can deal with different programming languages. The following is a short description of the tools we use in our experimentation.

To instrument our Java-based subjects, we use *Eclipse Test and Performance Tools Platform Project (TPTP)* version 4.4.2. The TPTP Project provides an open platform, supplying powerful frameworks and services for program monitoring, including test editing and execution, tracing and profiling, and log analysis. TPTP supports a broad spectrum of computing programs, including embedded, standalone, enterprise, and high-performance.

In this study, we use TPTP's tracing framework, which is called *Probekit*. Probekit allows developers to write fragments of Java code (called probes), that can be invoked at pre-specified points in the execution of a Java class to collect run time data about the program. Probekit offers various injection points that developers can use for their probes, including: Method entry, method exit, catch/finally blocks, class loading, etc. One common use of Probekit is to trace method invocations. Figure 6.3 illustrates a sample probe to print out on every method entry and exit, the method's name and parameters, process ID and Thread ID.

Probekit also provides a filtering mechanism that allows the user to determine which packages, classes or methods must or must not be instrumented. This helps reduce the size of the generated traces. Figure 6.4 illustrates filters of our sample probe. As illustrated in this figure, Java libraries such as "Sun.*", "java.*" are filtered out and thus no records from these packages are expected in the resulting traces.

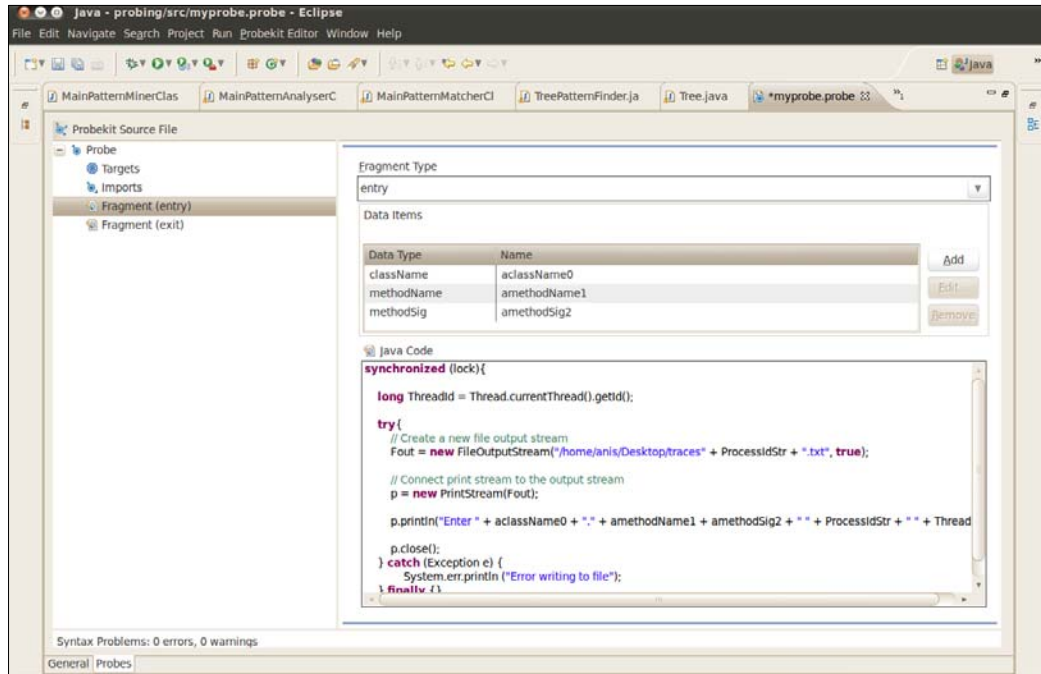


Figure 6.3: Sample probe created with Probekit editor

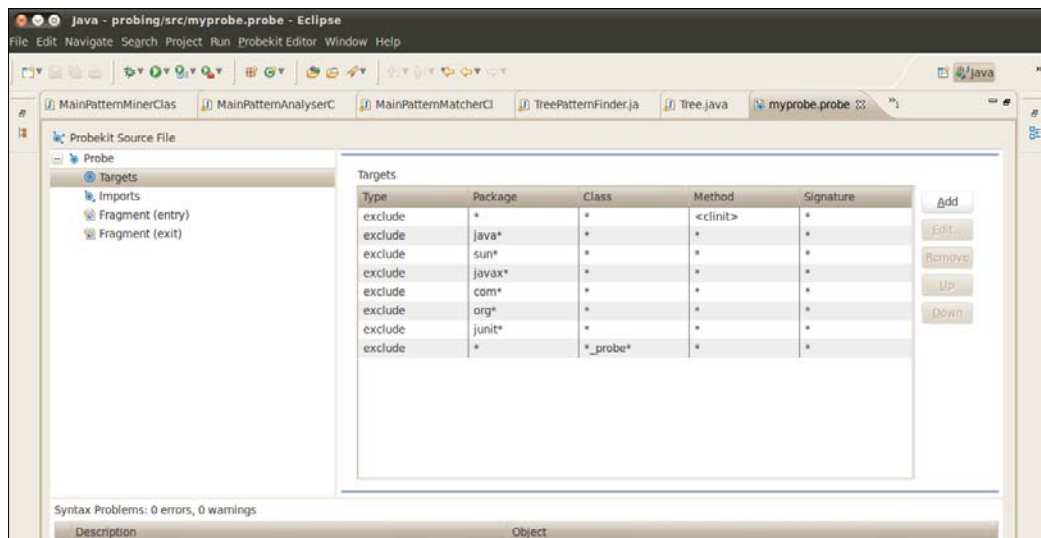


Figure 6.4: Filters associated with sample probe

To instrument our C-based subjects we use one of the features of the GNU GCC compiler, which enables tracing and profiling. GCC's `-finstrument-functions` option generates instrumentation calls for entry and exit to functions. A function may be given the attribute `no_instrument_function`, in which case the instrumentation will not be done. This can be used, for example, for the profiling functions themselves, high-priority interrupt routines, any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory), and any functions that the debugger does not want to instrument.

6.5 Localizing Defects in *FIXImulator*

In this section we show in detail how our defect localization tool can localize the defects that are embedded in our first subject program, the *FIXImulator*.

6.5.1 Test Cases and Test Oracles

To exercise *FIXImulator*, we need a suite of test cases. Our collaborator, the SWI company, provided a suite of test cases and their corresponding test scripts. The test cases in this suite are categorized into two groups “Session-Level” and “Program-Level”, where Program-Level test cases have further been categorized into “Pre-Trade”, “Trade”, and “Post-Trade”. In this case study we use the Trade test cases, which describe messages exchanged between the buy side (e.g., *Banzai*) and the sell side (e.g., *FIXImulator*) for trading activities such as placing a new order, canceling an order, etc. Figure 6.5 illustrates a sample Trade test case. As you see in this figure,

the buy side sends a new order to the sell side (line 1), the sell side acknowledges the order (line 2), fills the order in a number of steps (lines 3-4), and sends back a Done for Day (DFD) message (line 5).

Time	Message Received	Message Sent	Exec Type	Ord Status	Exec Trans Type	Order Qty	Cum Qty	Leaves Qty	Last Shares	Comment
D2.2										
1	New Order (X)					10000				
2		Execution (X)	New	New	New	10000	0	10000	0	
3		Execution (X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution of 2000
4		Execution (X)	Partial Fill	Partially Filled	New	10000	3000	7000	1000	Execution of 1000
5		Execution (X)	Done for Day	Done for Day	New	10000	3000	0	0	Assuming day order. See other examples which cover GT orders

Figure 6.5: Sample test case

FIXImulator's test cases not only identify what needs to be done in a testing scenario (e.g., sending a new order) but also indicate what is expected to be observed in a successful run (e.g., an acknowledge message, fill messages and a DFD message). In this sense, a test case also acts as a test oracle, indicating when a test passes or fails.

6.5.2 Tracing

Tracing is the first step of our defect localization technique. Since *FIXImulator* is a Java-based program, we use TPTP Probekit to instrument it. Then we run our test suite to collect the execution traces.

There are a number of ways one can run a test case. In *manual* testing, we follow the steps defined in a test case's scenario. In this approach, we use the buy-side

program, *Banzai*, to send trading orders to *FIXImulator*. Figure 6.6 is a snapshot of *Banzai* sending a buy request to *FIXImulator*. Figure 6.7 illustrates how *FIXImulator* receives the order and responds to it. As you see, *FIXImulator* displays the messages it sends/receives in a message flow table.

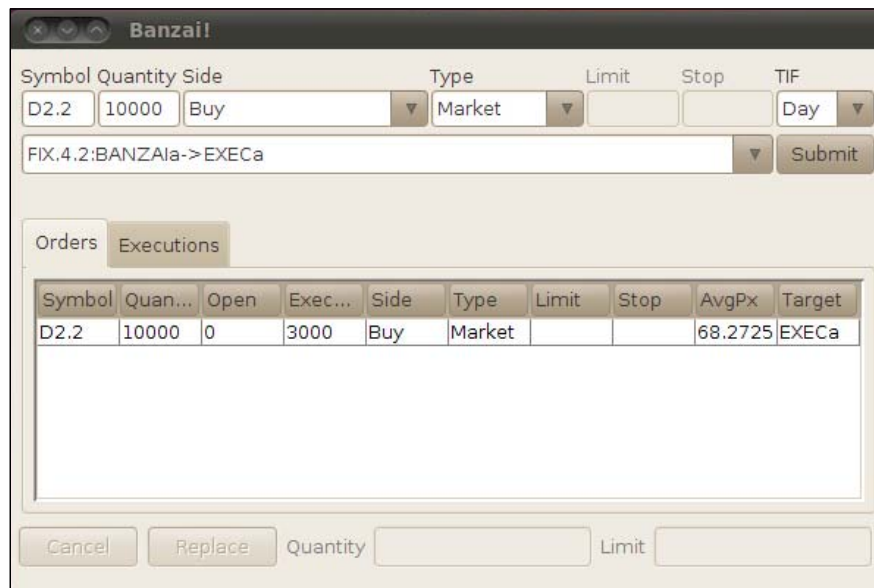


Figure 6.6: *Banzai* sending a new buy order to *FIXImulator*

In *automatic* testing, we use JUnit (JUnit, n.d.) to automatically execute scripts provided for each test case in the test suite. JUnit is a unit testing framework for the Java programming language which tests units of source code, i.e., sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, to determine if they are fit for use.

When running a test script, JUnit provides a report, including the pass/fail result and execution details. Figure 6.8 illustrates the execution of a sample test script using JUnit, where the test case has failed, because wrong message type is received.

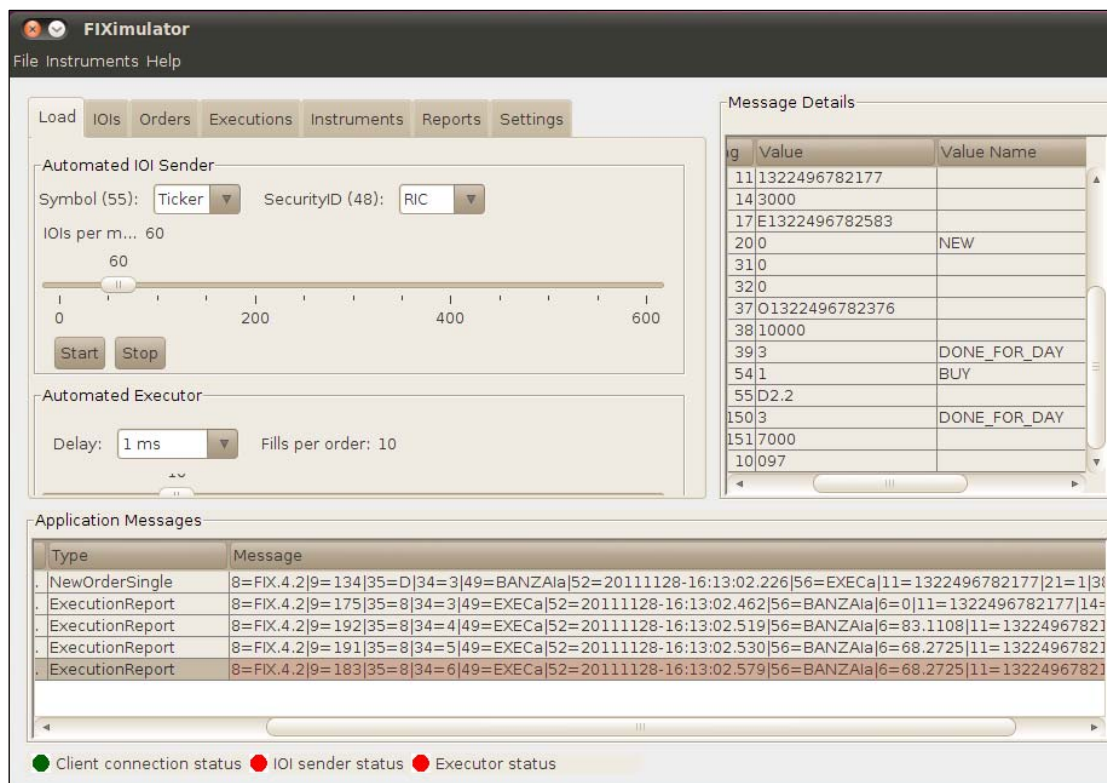


Figure 6.7: *FIXimulator* receiving a new buy order from *Banzai* and sending a number of messages

```

anis@anis-desktop: ~/anis/quickfixj/core
File Edit View Terminal Help
[junit] INFO: MINA session created for FIX.4.2:TW->ISLD: local=/127.0.0.1:50313, class org.apache.mina.common.nio.NioSession
alhost/127.0.0.1:9630
[junit] 10-Aug-2012 9:20:33 AM quickfix.test.acceptance.PrintComment run
[junit] INFO: # SEND: Market order of BNS for 10000 shares
[junit] 10-Aug-2012 9:20:34 AM quickfix.test.acceptance.PrintComment run
[junit] INFO: # EXPECT: Execution report - Acknowledge order
[junit] 10-Aug-2012 9:20:36 AM quickfix.test.acceptance.PrintComment run
[junit] INFO: # EXPECT: Execution report - Partial fill
[junit] -----
[junit]
[junit] Testcase: fix42/D2_2_DoneforDay.def took 60.069 sec
[junit] FAILED
[junit] wrong msg type expected:<[8]> but was:<[0]>
[junit] junit.framework.ComparisonFailure: wrong msg type expected:<[8]> but was:<[0]>
[junit]     at quickfix.test.acceptance.ExpectMessageStep.assertMessageEqual(ExpectMessageStep.java:93)
[junit]     at quickfix.test.acceptance.ExpectMessageStep.run(ExpectMessageStep.java:93)
[junit]     at quickfix.test.acceptance.AcceptanceTestSuite$AcceptanceTest.run(AcceptanceTestSuite.java:24)
[junit]     at junit.extensions.TestDecorator.basicRun(TestDecorator.java:24)
[junit]     at junit.extensions.TestSetup$1.protect(TestSetup.java:23)
[junit]     at junit.extensions.TestSetup.run(TestSetup.java:27)
[junit]
[junit] Test quickfix.test.acceptance.AcceptanceTestSuite FAILED

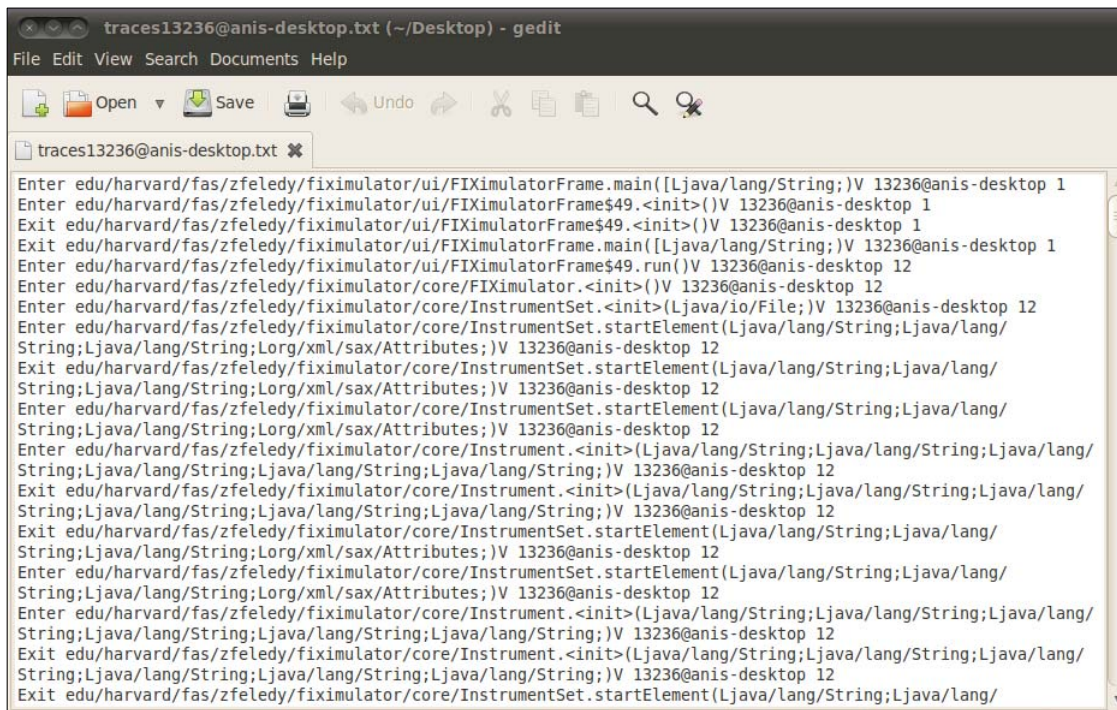
test.acceptance:
BUILD SUCCESSFUL
Total time: 1 minute 13 seconds
anis@anis-desktop:~/anis/quickfixj/core$

```

Figure 6.8: Running a test script using JUnit

Running a test case on the instrumented program produces dynamic execution traces. Figure 6.9 illustrates an excerpt of a sample trace file which indicates method entries and exits. Each line in this trace is composed of the keyword Enter or Exit, a method's class path, name and parameters, and ID of the process and thread running the method.

Note that our defect-localization technique requires that we execute every program several times and ensure that there is a sufficient number of examples for correct executions. This is necessary since we focus on occasional failures, i.e., failures whose occurrence depends on input data, random components or non-deterministic thread interleaving. Through experience we have seen that decision coverage is effective to provide useful defect localization results. However, even if the number of test cases are not big enough, we still get useful insights into the fault. Appendix A identifies



```
traces13236@anis-desktop.txt (~/Desktop) - gedit
File Edit View Search Documents Help
Open Save Undo
traces13236@anis-desktop.txt x
Enter edu.harvard.fas.zfeledy.fiximulator.ui.FIXimulatorFrame.main([Ljava/lang/String;)V 13236@anis-desktop 1
Enter edu.harvard.fas.zfeledy.fiximulator.ui.FIXimulatorFrame$49.<init>()V 13236@anis-desktop 1
Exit edu.harvard.fas.zfeledy.fiximulator.ui.FIXimulatorFrame$49.<init>()V 13236@anis-desktop 1
Exit edu.harvard.fas.zfeledy.fiximulator.ui.FIXimulatorFrame.main([Ljava/lang/String;)V 13236@anis-desktop 1
Enter edu.harvard.fas.zfeledy.fiximulator.ui.FIXimulatorFrame$49.run()V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.FIXimulator.<init>()V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.<init>(Ljava/io/File;)V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.startElement(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Lorg/xml/sax/Attributes;)V 13236@anis-desktop 12
Exit edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.startElement(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Lorg/xml/sax/Attributes;)V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.startElement(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Lorg/xml/sax/Attributes;)V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.Instrument.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V 13236@anis-desktop 12
Exit edu.harvard.fas.zfeledy.fiximulator.core.Instrument.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V 13236@anis-desktop 12
Exit edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.startElement(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Lorg/xml/sax/Attributes;)V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.startElement(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Lorg/xml/sax/Attributes;)V 13236@anis-desktop 12
Enter edu.harvard.fas.zfeledy.fiximulator.core.Instrument.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V 13236@anis-desktop 12
Exit edu.harvard.fas.zfeledy.fiximulator.core.Instrument.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V 13236@anis-desktop 12
Exit edu.harvard.fas.zfeledy.fiximulator.core.InstrumentSet.startElement(Ljava/lang/String;Ljava/lang/
```

Figure 6.9: Excerpt of a sample trace file

risks associated with insufficient number of test cases.

6.5.3 Pattern Mining

Upon collecting the traces, we run our tool to build dynamic call trees, reduce them, and mine frequent subtrees.

Figure 6.10 illustrates excerpts of an XML file that holds the discovered patterns. In this experiment, we ran 37 test cases. With the assumption of *Minimum Height threshold = 2* and *Minimum Frequency threshold = 2*, we mined 249 patterns. This means that we mined subtrees as small as a parent and its children, if the subtree is observed in at least two call trees. The XML file indicates the total pattern count. It also indicates for each pattern, the pattern's identifying number, its support set and the method calls representing its call tree (since a pattern is a dynamic call tree).



```

--<Patterns count="249">
--<Pattern id="1" supportSet="D27.1-manual.txt, D9.2.txt, D6.1.txt, D4.1.txt, D10.4.txt, D28.1.txt, D8.1.txt, D6.2.txt, D7.1.txt, D28.1-20000-manual.txt, D8.3.txt, D9.1.txt, D3.3.txt, D1.3.txt, D10.1.txt, D5.3.txt, D8.2.txt, D5.2.txt, D7.2.txt, D2.2.txt, D1.1.txt, D3.4.txt, D28.1-manual.txt, D4.4.txt, D10.3.txt, D8.4.txt, D2.1.txt, D7.4.txt, D6.4.txt, D3.2.txt, D1.2.txt, D4.2.txt, D27.1.txt, D6.3.txt, D10.2.txt, D27.1-20000-manual.txt, D3.1.txt, " methodCount="2">
--<method methodName="dummy">
--<method methodName="quickfix/mina/CompositeIoFilterChainBuilder.buildFilterChain(Lorg/apache/mina/common/IOFilterChain;V)/>
--</method>
--</Pattern>
--<Pattern id="2" supportSet="D27.1-manual.txt, D9.2.txt, D6.1.txt, D4.1.txt, D10.4.txt, D28.1.txt, D7.1.txt, D6.2.txt, D8.1.txt, D28.1-20000-manual.txt, D8.3.txt, D9.1.txt, D3.3.txt, D1.3.txt, D10.1.txt, D5.3.txt, D8.2.txt, D5.2.txt, D7.2.txt, D2.2.txt, D1.1.txt, D3.4.txt, D28.1-manual.txt, D4.4.txt, D10.3.txt, D8.4.txt, D2.1.txt, D7.4.txt, D6.4.txt, D3.2.txt, D1.2.txt, D4.2.txt, D27.1.txt, D6.3.txt, D10.2.txt, D27.1-20000-manual.txt, D3.1.txt, " methodCount="2">
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/Execution.<init>(Ledu/harvard/fas/zfeledy/fiximulator/core/Order;V)/>
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/Execution.generateID()Ljava/lang/String;"/>
--</method>
--</Pattern>
--<Pattern id="3" supportSet="D5.2.txt, D2.2.txt, D7.2.txt, D27.1-manual.txt, D9.2.txt, D28.1-manual.txt, D4.4.txt, D10.4.txt, D10.3.txt, D8.4.txt, D28.1.txt, D7.4.txt, D6.4.txt, D4.2.txt, D28.1-20000-manual.txt, D8.3.txt, D27.1.txt, D10.2.txt, D5.3.txt, D27.1-20000-manual.txt, D8.2.txt, " methodCount="2">
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication$Executor.stopExecutor()V">
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.access$502(Ledu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication;Z)Z"/>
--</method>
--</Pattern>
--<Pattern id="4" supportSet="D7.2.txt, D9.2.txt, D8.3.txt, D10.3.txt, D10.4.txt, D8.4.txt, D10.2.txt, D8.2.txt, D7.4.txt, " methodCount="2">
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication.setNewExecutorPartials(Ljava/lang/Double;V)/>
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/FIXimulatorApplication$Executor.setPartials(Ljava/lang/Double;V)/>
--</method>
--</Pattern>
--<Pattern id="5" supportSet="D5.2.txt, D7.2.txt, D2.2.txt, D27.1-manual.txt, D9.2.txt, D28.1-manual.txt, D4.4.txt, D10.4.txt, D10.3.txt, D8.4.txt, D28.1.txt, D7.4.txt, D6.4.txt, D4.2.txt, D28.1-20000-manual.txt, D8.3.txt, D27.1.txt, D1.3.txt, D10.2.txt, D5.3.txt, D27.1-20000-manual.txt, D8.2.txt, " methodCount="5">
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/InstrumentSet.getInstrument(Ljava/lang/String;)Ledu/harvard/fas/zfeledy/fiximulator/core/Instrument;"/>
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/Instrument.getTicker()Ljava/lang/String;"/>
--<method methodName="edu/harvard/fas/zfeledy/fiximulator/core/Instrument.getSeed()Ljava/lang/String;"/>

```

Figure 6.10: XML file holding discovered patterns

To analyze the stability of pattern mining (i.e., to see if more executions would lead to a significant change in the number of patterns discovered), we selected different subsets of test cases from our test suite and performed pattern mining. Figure 6.11 illustrates dependency of the total pattern count on the number of test cases.

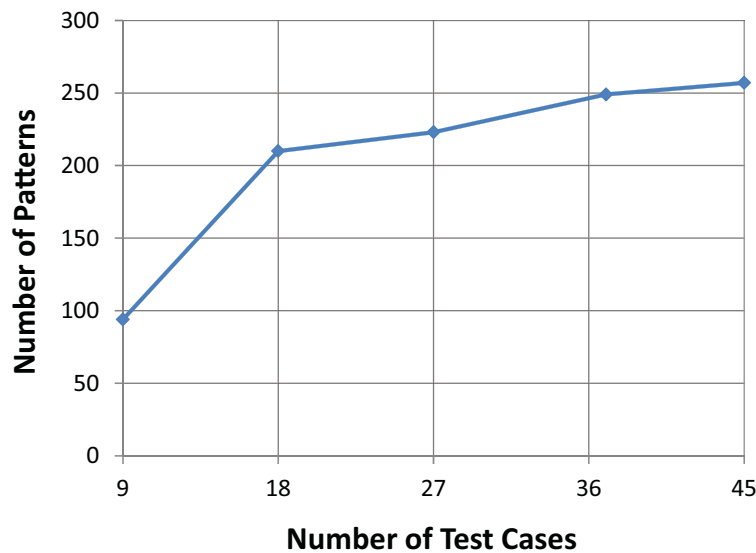


Figure 6.11: Relation between total pattern count and number of test cases

As you see, there is a non-linear relationship between the number of patterns and the number of test cases. So, increasing the number of test cases would not drastically change the number of patterns. In the worst case (assuming path coverage), the number of discovered patterns is bound by the number of paths in the call graph. However, path coverage is not realistic and we have seen through experience that decision coverage is good enough to provide good defect localization results. In this case, the number of patterns should not be greater than the total number of branches in the code.

6.5.4 Pattern Analysis

The first step in pattern analysis is to identify the mapping between tests and features (i.e., test-feature mapping). Since *FIXImulator*'s test cases describe messages communicated between buy and sell parties, we consider messages as features. In this context, a feature is defined as “communication of a particular message to/from *FIXImulator*” and a test case describes a scenario in terms of what messages are communicated. As a consequence, the number of features in the *FIXImulator* program is equal to the number of distinct message types in FIX. Table 6.5.4 presents a number of message types in FIX4.2. A complete list of message types in FIX is presented in the FIX protocol website (FIXMessages, n.d.).

Table 6.5: FIX Protocol Messages

Message Type	Sub-type
1 OrderSingle (new order)	-
2 ExecutionReport	2.1 Rejected 2.2 New 2.3 Partial Fill 2.4 Fill 2.5 Done for Day 2.6 Pending Cancel 2.7 Cancelled 2.8 Restated 2.9 Pending Replace 2.10 Replace
3 OrderCancelRequest	-
4 OrderCancelReject	4.1 New 4.2 Partially Filled 4.3 Filled
5 OrderReplaceRequest	-

A test-feature map indicates if a test exercises a certain feature. In the context of this experiment, a message type is mapped to a test case if the test case includes at

least one message of the specified type. Table 6.5.4 illustrates the test-feature map for 37 test cases and 20 features. In this table entries starting with letter “T” are FIX trade test cases and those starting with letter “F” are features. The number following letter “F” identifies the type of the message communicated. For example *F2.1* represents the communication of a “Rejected” execution report message, to/from *FIXImulator*.

The second step in pattern analysis is to identify the target feature (i.e., failing) feature. As an example, assume that test case *T5* fails and JUnit reports that the done for day (DFD) message contains some invalid values (e.g., field number 150 of the message is expected to be equal to three but is empty). From this report, we know that feature *F2.5* (i.e., communication of message Done for Day (DFD) to/from *FIXImulator*) has failed. Also, assume that the defect only affects feature *F2.5*. In this scenario, *F2.5* is the target of pattern analysis.

In the first step of pattern analysis, identifying *f*-specific candidate patterns, we use the test-pattern mapping provided by the *Pattern Mining Engine* and the test-feature mapping we created in the previous step to find patterns that are only observed in tests exercising feature *F2.5*. In this step, we found 34 out of 249 patterns as candidate feature-specific for *F2.5*. Next, we use the method call frequencies provided by *Trace Manager* to generate a sorted list of candidate patterns, where patterns are ranked according to their relevance to the failing feature. To do this ranking we use formulae *DS2* and *DS1* and the pattern ranking mechanism introduced in Section 5.5.4. Figure 6.12 illustrates this list for *F2.5*, where the numbers are pattern identifiers.

In the next step, assuming that the *rootsToPick* threshold is equal to five, we pick

Table 6.6: Mapping test cases to features they exercise

Test Cases	Features																				
	F1	F2.1	F2.2	F2.3.1	F2.3.2	F2.3.3	F2.4.1	F2.4.2	F2.5	F2.6	F2.7	F2.8	F2.9	F2.10.1	F2.10.2	F3	F4.1	F4.2	F4.3	F5	
T1	✓																				
T2	✓	✓																			
T3	✓		✓	✓			✓														
T4	✓	✓																			
T5	✓		✓	✓					✓												
T6	✓	✓																			
T7	✓		✓	✓																	
T8	✓		✓							✓							✓	✓			
T9	✓		✓							✓	✓						✓				
T10	✓	✓																			
T11	✓		✓	✓													✓				
T12	✓		✓	✓	✓					✓	✓						✓		✓		
T13	✓		✓	✓							✓						✓		✓		
T14	✓		✓	✓				✓		✓							✓				
T15	✓	✓																			
T16	✓		✓															✓			✓
T17	✓		✓										✓				✓				✓
T18	✓		✓	✓									✓	✓							✓
T19	✓	✓																			
T20	✓		✓	✓														✓			✓
T21	✓		✓	✓		✓	✓						✓		✓						✓
T22	✓	✓																			✓
T23	✓		✓				✓													✓	✓
T24	✓		✓				✓						✓							✓	✓
T25	✓		✓				✓						✓		✓						✓
T26	✓	✓													✓						✓
T27	✓		✓	✓			✓													✓	✓
T28	✓	✓																			
T29	✓		✓	✓														✓			✓
T30	✓		✓	✓		✓							✓					✓			✓
T31	✓		✓	✓		✓	✓						✓		✓				✓		✓
T32	✓		✓	✓					✓			✓									
T33	✓		✓	✓					✓			✓									
T34	✓		✓	✓					✓			✓									
T35	✓		✓	✓			✓		✓			✓									
T36	✓		✓	✓			✓		✓			✓									
T37	✓		✓	✓			✓		✓			✓									

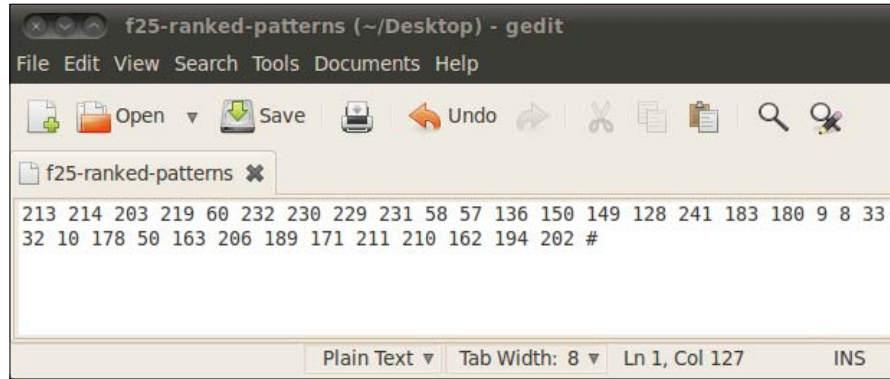


Figure 6.12: Ranked list of candidate feature-specific patterns for feature $F2.5$

patterns with at most five different roots to be feature-specific. Figure 6.13 illustrates the first ranked feature-specific pattern (pattern number 213) for feature $F2.5$. As you see, the root of this pattern is method `dfd` from the `FIXimulatorApplication` class, which based on its name, seems to be relevant.

6.5.5 Defect Localization

In this step, we compare call tree of the failed execution with each feature-specific pattern, found in the previous step, to identify new and missing method calls. Then, we sort the pattern-match pairs using *parentDiffSize* and *diffSize* values. Figure 6.14 illustrates the sorted list, which includes the comparison results. As you see, the third entry in this report indicates that method `dfd` is missing from method `onMessage`. Verifying the results, this actually was the true reason behind the failure.

The final step is to rank methods according to their likelihood of being responsible for the failure, which is built from the sorted list of pattern-match pairs. Table 6.5.5 presents defect localization results for 29 defective versions of `FIXimulator`. The

```

file:///media/wi...ces/patterns.xml
</feature>
--<Pattern id="213" supportSet="D27.1-manual.txt, D27.1.txt, D27.1-20000-manual.txt, " methodCount="650">
--<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.dfd(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV">
--<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.<init>(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV">
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.generateID(ILjava/lang/String;"/>
</method>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Order.setStatus(CJV"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.setExecType(CJV"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.setExecTranType(CJV"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Order.getOpenID"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.setLeavesQty(DJV"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Order.getTif(ILjava/lang/String;"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Order.getExecutedID"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.setCumQty(DJV"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Order.getAvgPx(DJV"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.setAvgPx(DJV"/>
--<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator
/core/Execution;JV">
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.getOrder(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;"/>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Order.getID(ILjava/lang/String;"/>
--<method methodName="quickfix/field/OrderID.<init>(Ljava/lang/String;JV">
--<method methodName="quickfix/StringField.<init>(ILjava/lang/String;JV">
<method methodName="quickfix/Field.<init>(ILjava/lang/Object;JV"/>
</method>
</method>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.getID(ILjava/lang/String;"/>
--<method methodName="quickfix/field/ExecID.<init>(Ljava/lang/String;JV">
--<method methodName="quickfix/StringField.<init>(ILjava/lang/String;JV">
<method methodName="quickfix/Field.<init>(ILjava/lang/Object;JV"/>
</method>
</method>
<method methodName="edu.harvard.fas.zfeledy.fiximulator/core/Execution.getFIXExecTranType(C"/>
--<method methodName="quickfix/field/ExecTransType.<init>(CJV">

```

Figure 6.13: Feature-specific pattern for feature $F2.5$, display of message done for day

```

---- Approach #1 - sort all:
*** pattern p60 (root=quickfix/field/ExecRestatementReason.<init>(I)V)
RNF: root of the pattern not found.
Expecting: quickfix/field/ExecRestatementReason.<init>(I)V
Prospective callers: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)

*** pattern p219 (root=edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.restate(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV)
RNF: root of the pattern not found.
Expecting: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.restate(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV)
Prospective callers: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.onMessage(Lquickfix/fix42/NewOrderSingle;Lquickfix/SessionID;JV)

*** pattern p213 (root=edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.dfd(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV)
RNF: root of the pattern not found.
Expecting: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.dfd(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV)
Prospective callers: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.onMessage(Lquickfix/fix42/NewOrderSingle;Lquickfix/SessionID;JV)

*** pattern p214 (root=edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.dfd(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV)
RNF: root of the pattern not found.
Expecting: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.dfd(Ledu.harvard.fas.zfeledy.fiximulator/core/Order;JV)
Prospective callers: edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.onMessage(Lquickfix/fix42/NewOrderSingle;Lquickfix/SessionID;JV)

*** pattern p203 (root=edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)
Location: 6053 - parentDiffSize: 1, diffSize: 10
missing< edu.harvard.fas.zfeledy.fiximulator/core/Order.getQuantity()D
from edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)
missing< edu.harvard.fas.zfeledy.fiximulator/core/Execution.getDayOrderQty()D
from edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)
missing< quickfix/field/DayOrderQty.<init>(D)V
from edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)
missing< quickfix/fix42/ExecutionReport.set(Lquickfix/field/DayOrderQty;JV)
from edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)
missing< edu.harvard.fas.zfeledy.fiximulator/core/Execution.getDayCumQty()D
from edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)
missing< quickfix/field/DayCumQty.<init>(D)V
from edu.harvard.fas.zfeledy.fiximulator/core/FIXimulatorApplication.sendExecution(Ledu.harvard.fas.zfeledy.fiximulator/core/Execution;JV)

```

Figure 6.14: Results of comparing $F2.5$ -specific patterns with call tree of failing execution

numbers in this table represent *Position* and *Rank* of the defective method in the ranked list. The ranking is based on Approach 1 introduced in Section 5.5.4 (sorting all pattern-match pairs using *parentDiffSize*, where *diffSize* is used as a tie breaker). As you see, except for the defective version number one, the defective method is ranked within the top four (out of 669) methods in the *FIXImulator*. For version one, in the worst case, the defective method is still within the top 2.8% of the methods. This means that 97.2% of the methods need not be examined in the search for the defect ($Score(M) = 97.2\%$). These results correspond to 37 test cases, different defect types, and different failing features.

Table 6.7: Results of defect localization for FIXImulator

Versions	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15
Position (random case)	6	1	1	1	4	2	1	3	4	2	2	2	2	3	3
Rank (worst case)	19	2	1	1	4	2	1	3	4	2	2	2	2	3	3

Versions	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28	V29
Position (random case)	3	3	3	1	1	1	2	4	2	2	2	2	4	4
Rank (worst case)	3	3	3	2	1	1	3	4	2	2	2	2	4	4

6.6 Localizing Defects in *Weka*

Weka's defects were all introduced in `weka.classifiers.trees.DecisionStump` class, which implements a decision-tree-based classifier algorithm. Although all of the defective versions target a single feature (i.e., the `DecisionStump` classifier), in our experimentation we traced other features of the program as well. This includes other classifiers, clusterers, and association rule miners. Tracing other features helps to rule out common and omnipresent patterns.

Eichinger et al., (Eichinger *et al.*, 2010b) provide 90 test-input data to exercise *Weka*, leading to successful and unsuccessful runs. We used a subset of the test-input data to exercise different features of *Weka*. The reasons behind this are as follows: 1) We intended to show that even without proper code coverage we can still get reasonable results; 2) Many of the test-input data led to a failing run and we needed only one failing trace, which we chose randomly; 3) *Weka* is a big program and tracing some inputs was too slow; 4) The test-input data was originally intended to be used for exercising the `DecisionStump` class and thus did not have the correct format to be used to exercise other features.

We emphasize that, even though we knew the defective class, we instrumented all 19,395 methods of *Weka*, thus all of them were potential subjects of defect localization. To identify if a test passed or failed, we developed a program that compares output of a test applied on a defective version against output of the same test applied on the original (non-defective) version.

Table 6.6 provides results of defect localization in *Weka*. The results correspond to Approach 1 in ranking pattern-match pairs introduced in Section 5.5.4. As you see, in this experiment the defective method is always within the top seven ranked methods. Provided that *Weka* has about 19K methods, this corresponds to a $Score(M)$ of above 99.9%.

Table 6.8: Results of defect localization for Weka

Versions	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16
Position (random case)	6	6	4	4	4	4	3	5	7	2	6	3	3	3	4	5
Rank (worst case)	6	6	4	4	4	5	4	5	7	3	6	4	4	3	4	5

In all defective versions of Weka, except *V3* and *V10*, the reported numbers associate with the actual defective method (i.e., the root cause). For *V3* the reported results belong to the parent of the actual defective method (*distance to root cause* = 1). For *V10* the reported results belong to an ancestor of the actual defective method (*distance to root cause* = 3).

In this step we compare our results with Eichinger et al., (Eichinger *et al.*, 2010b). For the sake of comparison, we provide Eichinger et al.'s results in Table 6.6. This table presents results associated with six different approaches *E1* to *E6* presented in (Eichinger *et al.*, 2010b). Although, Eichinger et al.'s best results (corresponding to approach *E3*) are better than what we got, our results are still comparable (i.e., our numbers are in the range of numbers Eichinger et al. has published). Moreover, Eichinger et al. assumes multiple failing cases, and in this sense our approach is more powerful. As we discussed in Chapter 4, Eichinger et al.'s approach cannot deal with a single failing case as effectively.

Table 6.9: Eichinger et al.'s results for Weka (Eichinger *et al.*, 2010b)

Versions	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16
E1	3	3	1	3	2	2	12	3	1	1	2	2	1	2	1	3
E2	2	2	1	2	2	2	9	2	1	1	2	2	1	2	1	2
E3	1	1	1	2	6	1	1	1	3	5	1	1	1	1	1	1
E4	1	1	11	13	10	3	13	10	9	6	3	8	1	3	8	10
E5	1	1	4	5	4	2	7	3	5	4	2	4	1	2	3	3
E6	1	1	1	2	7	1	2	1	8	6	1	1	1	1	1	1

6.7 Localizing Defects in *NanoXML*

Do et al., (Do *et al.*, 2004) provided an extensive test suite for *NanoXML*, containing more than 200 test-inputs for each defective version of the code. *NanoXML* is a single-feature program. Therefore, we had to skip the pattern analysis step when looking for defects, because all mined patterns are considered relevant to the only feature of the program, and should be checked against the failing tree.

Table 6.7 provides the results of defect localization on *NanoXML*. In this table we report the number of methods atop the defective method. The provided results associate with approaches 1 and 3 in the ranking pattern-match pairs. In Approach 1, we sort all patterns base on *parentDiffSize* where *diffSize* is used as a tie breaker. In Approach 3, we sort all patterns based on Dallmeier et al.'s formula where *parentDiffSize* and *diffSize* are used as tie breakers.

Table 6.10: Results of defect localization for NanoXML

Versions	V1	V2	V3	V5	Average
Methods Atop - Approach 1	44	14	22	-	26.7
Methods Atop - Approach 3	1	14	3	-	6

As you see, Approach 1 does not provide very good results. Compared with the results we got for *NanoXML*, our results for bigger subjects such as *FIXImulator* and *Weka* are much better. This may be because we could take advantage of feature location to reduce the size of the search space in bigger subjects whereas for smaller (one-featured) subjects this is not true. In one-featured applications, all patterns

are considered relevant, thus the pattern analysis part, which helps locating feature-specific patterns, is skipped. Another possible reason could be that Approach 1 is not really a good fit for this program, and perhaps not a good fit in general. As you see, the results improved with Approach 3. However, whether Approach 3 is better in general requires further investigation, which is left for future work.

In this stage, we compare our results with Dallmeier et al.'s (Dallmeier *et al.*, 2005). One should note that Dallmeier et al., provide a ranked list of suspicious classes. Based on their results, on average there are 2.22 to 3.69 classes atop of the defective class in the ranked list. *NanoXML* has 10.1 methods per class on average. Assuming that the defective method is the first method investigated in the class (the best case), Dallmeier et al.'s results correspond to 22.4 to 37.2 methods atop of the defective method. In contrast, using our ranking approaches 1 and 3, our results correspond to the investigation of 26.7 and six methods on average, respectively. Thus, Approach 1 is within the range and Approach 3 improves over Dallmeier et al.'s results.

As mentioned before, no defect definition files were found for version four of the code. Therefore, we do not provide results related to *V4*. For *V5*, we could not locate the defects. The reason for that is as follows. *NanoXML* test cases consist of a test driver, which basically is a `main` method calling different *NanoXML* methods. Whenever a driver calls a specific *NanoXML* method (ancestor of the defective method), the test fails. Assume this ancestor is method `B`. So, in passing tests, `main` calls methods `A` and `C`, and in failing tests it calls `A` and `B`. In this case, no correct patterns can be attributed to `B`, since no correct executions include method `B`. However, the existence of method `B` in the failing execution while it is not seen in any correct executions can

be a good clue leading us to the actual location of the defect. This can simply be checked by looking for methods which appear in the failing execution only. Yet, this is not implemented in the current version of our tool, and we are only depending on correct execution patterns to find the location of defects. In this case if `main` calling `A` and `C` happens so frequently that we capture it as a frequent pattern, comparing this pattern with the failing execution will point us to the missing `B`. Unfortunately, this has not been the case in *V5*. The aforementioned problem is one of the risks of using our tool, which is discussed in Appendix A (Risk *v* in A.3). The suggested solution for this problem and any other possible mitigation require further investigation, which is left for future work.

6.8 Localizing Defects in *Print_tokens*

The test-input required to trace *Print_tokens* is provided by Siemens researchers (Hutchins *et al.*, 1994). Like *NanoXML*, *Print_tokens* also has only one feature (i.e., tokenizing the input), therefore no pattern analysis is performed and all mined patterns are considered relevant. Table 6.8 provides the results for *Print_tokens*. The results correspond to Approach 1 in ranking pattern-match pairs. As you see, the defective method is within the top six ranked methods. As *Print_tokens* has 18 methods, this corresponds to a $Score(M)$ of greater than 68%.

Yu *et al.*, (Yu *et al.*, 2011) provided an evaluation of their technique (Loupe) as well as a number of other statement-level spectra-based techniques using the Siemens suite. Figure 6.15 is borrowed from (Yu *et al.*, 2011) and illustrates their evaluation results. There are seven graphs associated with seven defect localization techniques: Loupe (Yu *et al.*, 2011), Tarantula (Jones and Harrold, 2005), Sober (Liu *et al.*, 2006),

Table 6.11: Results of defect localization for `print_tokens`

Versions	V1	V2	V3	V4	V5	V6	V7
Position (random case)	3	1	2	4	1	5	5
Rank (worst case)	5	4	4	5	6	5	5
Score	0.73	0.78	0.78	0.73	0.68	0.73	0.73

PPDG (Baah *et al.*, 2008), CrossTab (Wong *et al.*, 2008), BARINEL (Abreu *et al.*, 2009b), and avgSBD (Santelices *et al.*, 2009). The measure used for evaluation is $Score(S)$, which is defined similar to the $Score(M)$. A point (x, y) in a graph indicates that y percentage of defective versions have $score(S)$ of greater than or equal to x . Given a ranked list of suspicious statements S , $score(S)$ is computed as the percentage of statements in the ranked list that need not to be examined after reaching the defective statement, and is computed as:

$$Score(S) = \frac{\text{Total Number of Statements} - \text{Rank of the Defective Statement}}{\text{Total Number of Statements}} * 100\%.$$

We added the results of our experimentation with `print_tokens` to this figure, which corresponds to the magenta graph marked with stars. One should note that since we are dealing with methods, the score we compute is $Score(M)$ which is different from $Score(S)$ used by Yu *et al.* Since we are dealing with methods in this work, the score we compute is based on the number of methods, which is different from the statement-based score used by Yu *et al.* However, with the (unlikely) assumption that all methods have the same number of statements and that the defective statement is the last statement in the defective method (the worst case), there is a relationship between our definition of Score and the score used by Yu *et al.*, and so a rough comparison can be made.

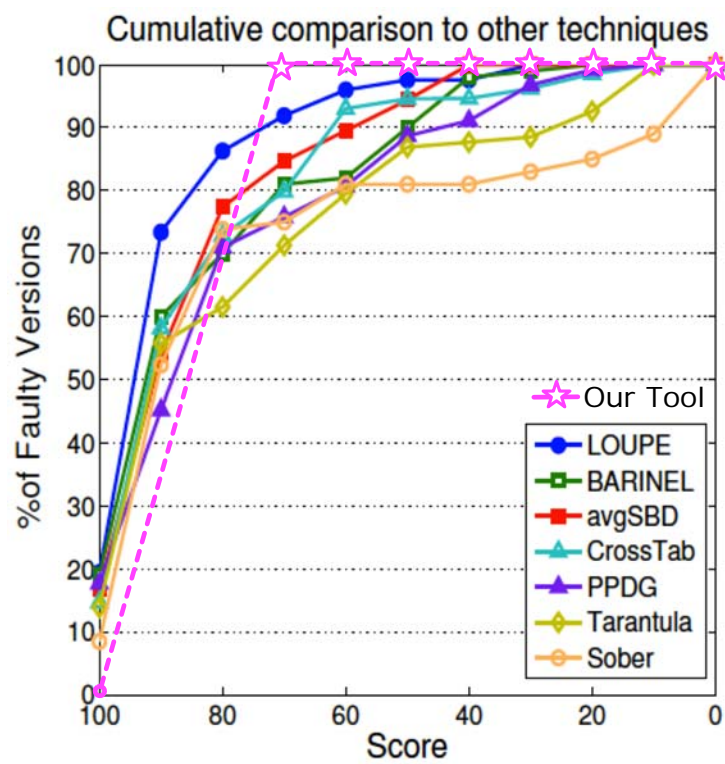


Figure 6.15: Comparison of a number of approaches using Siemens suite subject provided by Yu et al., (Yu *et al.*, 2011), with additional data points from our tool

There are a number of points to consider when comparing our results with Yu et al.'s:

- The *Print_tokens* program is very small (18 methods), thus a call-tree-based approach such as ours is not very helpful. Considering the definitions for $Score(M)$ and $Score(S)$, in the best case, when the defective method/statement is ranked first in the list, the corresponding score is less than 100% in both cases. However, since the number of methods is much smaller than the number of statements, the effect of a small change in the rank of the defective method/statement on computing the percentages is much greater for $Score(M)$ as compared to $Score(S)$. For example in a program with 10 methods and 100 statements, in the best case (if the defective method is ranked first in the list), the computed $Score(M)$ is equal to 90% which corresponds to having the defective statement at rank 10.
- The approaches compared in this diagram deal with different underlying assumptions and use different techniques for defect localization. For example, all of the presented approaches are spectra-based, requiring multiple instances of the failing execution. Dealing with one failing execution, our approach is more powerful than the other approaches in the diagram. On the other hand, our approach localizes defects in the granularity of a method, which is less precise than its counterparts, which deal with statements.
- The approaches compared in this diagram have different time and memory overheads. In general statement-level approaches need to deal with huge statement-level traces, which is a big issue in very large target programs. This is one of the reasons they target smaller programs such as the Siemens suite. In comparison, method-level approaches such as ours impose less tracing overheads and

are thus preferable for bigger programs.

- Yu et al. provided these results by performing experiments on all seven Siemens programs (about 120 defective versions), however, our results are based on *Print_tokens* (seven defective versions).

Considering the above discussion, the results provided by our technique are still comparable with other techniques presented in Figure 6.15. More experimentation with other programs in the *Siemens Suite* is needed to confirm this. However since small subjects are not targeted in this research, we left this for future work.

6.9 Discussion

This section provides a summary of the lessons we learned from applying our defect localization technique on different subject programs, and the challenges we encountered.

6.9.1 Lessons Learned

Dealing with different subject programs, types of defects, and settings, we learned a lot about defect localization in general as well as our proposed technique.

- Our experiments illustrated the applicability of the proposed defect localization technique. The proposed technique was shown to be more effective on bigger programs such as *FIXImulator* and *Weka*. However, there is a need to improve our pattern-match ranking mechanism to enhance the results. As shown in *NanoXML* experiment, better ranking mechanisms such as Approach 3 can improve the results. This needs to be verified with further experimentation.

- Although the proposed technique does not necessarily beat spectra-based approaches, it provides comparable results. When comparing the results one should consider that: 1) Compared to statement-level approaches, method-level approaches deal with relatively smaller traces and thus starting from methods and digging into a few potentially defective ones seems to be an advantage when dealing with very large target systems; 2) call-graph-based spectra-based approaches assume multiple failures of the same nature. The single-failure assumption in this work makes the analysis more challenging. Nevertheless, our technique was able to provide good clues to lead the programmer to the defective methods in various cases. We could verify this for the *FIXImulator* subject, by presenting our results to an expert in SWI company.
- We need more comprehensive experimentation to evaluate the usefulness of our technique. We need to analyze time and space overheads. We also need to analyze the stability of the results (i.e., to see whether the number of false positives change as a result of changing the number of test cases, which as we presented before changes the number of patterns).
- In a technique such as the one discussed in this thesis, the reported method is not necessarily the root cause of the failure. In such approaches, criteria such as “rank of the first relevant method”, “number/percentage of relevant methods that proceed the actual defective method in the ranked list” and “distance of the top ranked method to the root cause” can also be useful.
- In addition to finding the potential location of defects, the proposed technique

also identifies suspicious missing or additional method calls within the potentially defective method. This can lead to better understanding of the problem and a lower overall cost for debugging.

- Using details about the failure, such as what features or events are problematic, are good clues to narrow down the search space for finding defects. They narrow the scope of the search to the relevant code, instead of doing a blind search for all suspicious changes.
- Without pattern mining it is generally hard to determine which parts of a dynamic call tree are involved in the execution of the failing feature. Thus, to locate the problem one would need to examine a bigger search space. For example, one of our tests on *FIXImulator* created approximately 72K method calls which is a rather big search space to locate a defect. In contrast, a typical pattern in *FIXImulator* had an average of 161 method calls. Therefore, using a pattern or a few patterns representing the failing feature was wiser.

6.9.2 Challenges of Experimental Evaluation

The following is a list of challenges we faced during our experiments.

- Benchmarks used in the literature of defect localization are small programs and cannot really demonstrate the effectiveness of defect localization approaches. The diversity of the programs used by researchers in this area makes comparison challenging and inefficient. Moreover, there is no agreement on the measures researchers use to report their results. Measures such as score, as used by Yu et al., (Yu *et al.*, 2011), or averaging, as used by Dallmeier et al., (NanoXML, n.d.),

do not really convey enough information about the actual defect localization results. We believe that measures such as rank of the defective statement/method are more informative.

- In some cases it is not clear which version of the program has been used for experimentation by the researchers.
- In some cases, subject programs, their pre-requisite programs, their defective versions, test inputs that were exercised to raise the defects are not available for download.
- Subject programs or their pre-requisite programs cannot be installed or are hard to install. In some cases, such as *NanoXML*, there are errors in the compilation.
- Proper documentation is not available for subject programs or the provided documentation does not reflect the code. For example, there are fault matrices available for *NanoXML* and *Print_tokens* which are supposed to indicate which tests are failing, but are not accurate. In the case of *NanoXML*, defects of version four were not documented, so we could not verify our results.
- Lack of proper documentation in some cases results in difficulty understanding how one should run/instrument the target program and execute a test.
- In some cases, the provided test cases are not enough or do not cover certain parts of the code (e.g., only testing one feature).
- Many manual operations need to be done to trace the programs and verify the results by checking them against the actual defects.
- Tracing is time consuming.

Chapter 7

Conclusion and Future Work

In this thesis, we investigated call-graph-based defect localization and proposed a novel technique that incorporates tree-mining and tree-matching to provide hints as to where the defects may be located. In this research, we have considered a number of challenges. First, we assumed that the target systems to be analyzed are large multi-functional systems and that it is hard to construct scenarios that only execute a single feature of the system. This leads to the generation of huge traces (and extensive call graphs) which do not exclusively represent the feature of interest. Considering such call graphs as a whole leads to too many false positives. We incorporated feature location techniques to identify feature-specific sub graphs, thus reducing the code that needs to be searched for defects. Second, we assumed the availability of minimum information about a failure - a single failing scenario. This is different from most call-graph-based defect localization techniques which assume multiple failing cases and incorporate statistical formulations to identify code discriminating correct from failing runs. The following sections provide a summary of the proposed approach, some concluding remarks, and directions for future research.

7.1 Summary

The defect localization part of this thesis started with an introduction to the research problem. Then, we presented introductory information about dynamic analysis, defect localization, feature location and data mining. Next, we provided a survey of dynamic-analysis-based techniques for defect localization. We observed that call-graph-based defect localization, a fairly recent technique in this domain, is promising for localization of structure and frequency affecting defects. However, it could potentially be improved to consider systems with multiple functionalities where only one failing test case was provided. We consequently proposed a call-graph-mining and matching-based approach which dealt with the above-mentioned challenges. We then implemented our technique as a prototype defect localization tool and evaluated it using four target applications with known defects. Our experiments showed that the results provided by our technique are comparable with those presented in the literature, bearing in mind that we are dealing with a more challenging problem in the sense that we target bigger programs and assume the availability of a single failure case. The following section lists our contributions, the lessons we learned in this research and uses of the proposed technique.

7.2 Discussion

The major outcome of this research is a novel defect localization technique that is able to handle large multi-feature systems with as little information as a single failing test case, and still provide an informative report. As part of this research, we also provided the following contributions:

- We enhanced state-of-the-art feature location by introducing the notion of feature-specific patterns. Feature-specific patterns provide context to the conventional feature-location where methods in the code had been associated with the features of the system. They illustrate how a method behaves in terms of the way it calls other methods. Thus, they can help understand methods that are commonly used by a number of features, by identifying in which scenarios the shared methods are specific to a feature. In this process: i) we suggested new formulations for quantifying method-feature relevance. The new formulations favor methods that are not only specific to a feature but also fundamental to the execution of the feature (i.e., highly probable in an execution exercising the feature); ii) we considered non-determinism in feature execution and thus dealt with non-deterministic patterns which required more complex analysis; and iii) we devised a novel approach to analyze the specificity of patterns to features based on their internal methods.
- We examined the challenges of distributed execution tracing in the special case of Web-service-based systems and suggested an approach to aggregate the distributed traces.
- We devised an algorithm for mining frequent bottom-up subtrees which enhances and improves its existing counterparts.

Experimenting with our tool, we found out that:

- Call trees better represent defects compared with single methods. The information such as “method B is called (or method B is called from method A) in all faulty executions and no correct ones” is an important clue to localize the

source of a failure. However, it can lead to false negatives and false positives. For example, method **A** can call method **B** in all faulty executions simply because a certain functionality is exercised in faulty executions only. This leads to a false positive. Also, method **A** can call method **B** in what appears to be a random way and not considered relevant to the defect. However, this can be because the target (faulty) feature has a non-deterministic call graph and in the specific failing case at hand can point to the location of the defect (false negative). The lesson we learn here is that a single method call is not enough for us to make wise decisions with regard to the defect. Context of the method call is important. Context can be defined as surrounding method calls. In this thesis we used a tree of method calls as a context for a suspicious call. In this method, a relevant subtree (a subtree representing the execution of the target feature) is used to differentiate relevant from irrelevant method calls.

- Feature-location can improve defect-localization. The proposed defect localization technique takes advantage of feature location techniques. Among other benefits, feature location provides the ability to narrow the code we search for a defect. When one searches for a failures source, one only needs to consider only relevant sections of the code. For example if a failure happens in the user interface (UI) of a program it should be located in the UI's code (unless it is related to the communication between UI and some other section of the code). The first identifies dynamic call trees associated with different features of the system (including the failing feature). It then uses relevant subtrees to locate suspicious method calls. The incorporation of feature location and approximate pattern matching enhances the state of call graph based defect localization by

introducing a new approach to defect localization which requires less information about the system (it uses a single failing case as opposed to many failing cases that is used by similar approaches).

The proposed tool is useful for software engineers in the following areas:

- Defect Localization: to locate structure-affecting defects in a system's source code, which aids engineers in the process of root cause analysis and debugging.
- Feature Location: to understand how a feature is implemented which is useful in system comprehension, debugging, migration, maintenance, etc.
- Software Migration: to distinguish technology-related code from the core logic of a system. In general feature-specific patterns represent the core of a system and common patterns represent utilities and technology-related code. However, this requires more investigation.

All in all, following our experimentation with a number of real world subjects, we found the outcome of our work beneficial for program understanding and consequently defect localization and correction. The report generated by our tool provided useful knowledge about passing and failing executions of different features of the target system, if not pointing to the exact location of the defects. Such knowledge could assist the software support team to better understand and maintain the system.

7.3 Future Work

Based on our study in this research we have identified some possible future research directions as presented below.

Managing large traces. Software systems tend to produce huge dynamic traces which require special considerations to be handled. One enhancement to the current defect localization technique could be to incorporate available knowledge about a failure to reduce the overall size of the required traces (and consequently their associated dynamic call trees). For example, in the pattern mining phase of the proposed technique, one may use a subset of traces (those related to the failure) for mining. e.g., if the failure is related to systems interactions, we can use only test cases provided for that matter. Furthermore, irrelevant subsystems, classes, or methods can be omitted in the search for defects. Another approach is to break down large dynamic call trees into smaller subtrees and devise a proper algorithm for pattern mining.

Localizing other types of defects. In this work we focused on localizing structure-affecting defects in single-threaded programs. By keeping track of frequency of method calls and incorporating them in the approximate matching process, we can extend the proposed technique to handle frequency-affecting defects. Also, we are interested in investigating the idea of mining- and matching-based defect localization to manage control-graph affecting, data-flow affecting and concurrency defects. In this case, one should incorporate other graph representations of dynamic executions. For instance, communication graphs (Lucia and Ceze, 2009) can be used to localize concurrency defects. We have conducted an initial study of concurrency defect localization in Appendix C.

Providing feedback for testing. One of the main drawbacks of dynamic analysis is its incompleteness. Software testing's code coverage criteria provides a means to control the coverage of the test cases and mitigate this problem. Another possible solution is to indicate which parts of the static call graph of a system are not covered

by dynamic call graphs built from test cases and thus identify the need for more test cases. This can also be used to detect minimum number of test cases required to cover all code.

Maintaining patterns database. The proposed defect localization approach performs pattern mining as one of its main activities. When one finds and fixes a defect, one may change some of the dynamic call graphs and their corresponding patterns in the database. Therefore, to locate more defects one may need to change the patterns which involves running the pattern mining phase all over again. One improvement in this area could be to keep the patterns up-to-date by applying the fixes to relevant patterns to update them according to recent changes.

Dealing with multiple features, feature interaction. In this work we have assumed features to be independent. We also assume that one feature fails in the failing execution. However, other scenarios can be assumed: a failure during an execution of the feature can be related to other features executing before it (feature interaction), a defect can make multiple features to fail, there can be multiple instances of a feature where one of the instances fails. To make a powerful defect localization tool, one needs to consider all different scenarios.

Chapter 8

Request Replication: An Alternative to QoS-aware Service Selection

In this chapter we present our second research subject which is *designing a novel alternative strategy for service selection in SOA with the aim of satisfying the QoS requirements of a client in a more cost-efficient way*. The proposed technique, namely *Request Replication*, has also been published as a conference paper (Yousefi and Down, 2011a) and a technical report (Yousefi and Down, 2011b).

8.1 Introduction

Service Oriented Architecture (SOA) is an increasing trend in developing business applications. In this architecture, software functionality is represented as a set of services with well defined interfaces which can be reused to build various types of

applications. A service is published by a service provider (from now on, we will simply use “provider”) and used by one or more service clients (“client” for short). Research in web services includes many challenging areas such as service composition, quality of service (QoS) aware service selection, etc.

Service selection is the process of choosing a service implementation from a pool of previously published services in a way that the selected service satisfies a client’s functional and non-functional requirements. This process is done in two steps. In *Functional-based* service selection (Baltoni *et al.*, 2006; Klusch and Kapahnke, 2008), services matching a set of functionality requirements are retrieved. In *Non-functional based (or QoS-aware)* service selection (Tian *et al.*, 2004; Yang *et al.*, 2007), functionally equivalent services, discovered in the previous step, are ranked based on non-functional requirements of the client.

The non-functional requirements are soft constraints on non-functional properties (NFPs), including quality of service (performance, security, reliability, response time, call cost, etc.) and Context (location, intention, client name, provider details, etc.). In the process of QoS-aware service selection, a client submits to the service selection engine a set of non-functional requirements. Furthermore, many service providers advertise functional and non-functional capabilities of their services at the time of publication. Hence the service selection engine can match requirements of a client against advertised capabilities of services. In SOA, there are two general approaches for satisfying QoS requirements. In the first approach, a client chooses from the pool of available services, the service which best matches non-functional requirements. An alternative to this approach is QoS Negotiation, in which the client negotiates with the provider to reach an agreement with regard to the non-functional requirements.

Non-functional properties of services, such as response time, are stochastic in nature. The dynamics of the environment in which a service is deployed, such as network-related delays and server congestion, can result in high variability in service non-functional properties. This yields two outcomes: On the negative side, the selected service may for a particular service invocation have response time that significantly exceeds the average value advertised. On the positive side, one could take advantage of the inherent variability in non-functional properties of a service to propose alternative service selection strategies.

In this chapter, we present a novel alternative strategy, namely *Request Replication*, to satisfy the QoS requirements of a client in a more cost-efficient way by taking advantage of the existing high variance in NFPs. Unlike conventional QoS-aware service selection, in *Request Replication* we choose from available services a set of independent low-cost and low-quality services in a way that their combination provides the required QoS. Throughout this work, we specifically consider response time as a representative of performance related NFPs. We concurrently send a request to a set of services, take the fastest response, and discard the remaining requests.

The term “replication” has been used in other contexts in the literature of QoS-aware service selection. For example, *Service Replication* is a mechanism providers use to guarantee their quality of service obligations in their service level agreements (SLAs) (You *et al.*, 2009). Also, the idea of sending a request to multiple functionally equivalent services followed by a voting mechanisms is used in the context of fault tolerance (Looker *et al.*, 2005; Salatge and Fabre, 2007; Zheng and Lyu, 2009).

In this work, we enhance the state of QoS-aware service selection in SOA from the client’s perspective. The contributions of this work are:

1. We provide an alternative strategy to conventional “QoS-aware service selection” in SOA, which has the potential to allow clients to have higher quality services with less cost. In this method, a client can build better quality services from low cost, low quality ones.
2. We present a number of recommendations about service advertisements for performance related NFPs, and specifically service response time. We believe that the advertisements should provide enough information about non-functional properties of a service to enable clients to make better decisions with regard to service selection. For this reason, we believe that clients should know the distribution of NFPs, or a sufficient number of parameters to estimate the distribution.

The rest of this chapter is organized as follows: Section 8.2 presents related concepts in the domain of service selection. Section 8.3 discusses related work in the literature of QoS-aware service selection in SOA. Section 8.4 presents the proposed “Request Replication” strategy. Section 8.5 provides a number of recommendations for service advertisements. Section 8.6 concludes this chapter.

8.2 Service Selection in SOA

In this section, we discuss the QoS model of SOA. Specifically, we explain the basics of QoS-aware service selection and negotiation as well as the nature of QoS advertisements.

8.2.1 QoS-aware Service Selection

Services in SOA are functional units, wrapped in well defined interfaces which are published for others to use. At the time of publication, a provider registers a service with a registry by providing information about the functionality and interface of the service along with its non-functional properties. QoS-aware service selection is the process of choosing a service implementation from a pool of previously published services in such a way that the selected service satisfies a set of functional and non-functional requirements. The basic building blocks for any service selection approach are discussed below and in (Sathya *et al.*, 2010).

Client Service Requirements

To find an appropriate service, a client submits to the service selection mediator a set of requirements along with their request. The requirements may involve both functional and non-functional aspects which need to be satisfied by the candidate service.

Provider Service Advertisements

The services offered by providers are concerned about functional and non-functional aspects. The providers thus specify both functional and non-functional properties of services in what is called a “service offering” or “service advertisement”. The functional properties include service parameters, messages, behavior and operation logic. The non-functional properties include QoS (security, reliability, response time, call cost, etc.) and Context (location, intention, client name, provider details, etc.). The non-functional properties are usually defined using a QoS ontology.

Service Selection Process

The service selection process involves finding a match for a client's requirements, among available service advertisements. In a basic form, service selection provides the best match for the client's requirements. In a more general form, service selection provides a ranking of the available services with regard to the client's requirements. Many service selection techniques and algorithms are proposed in the literature. These techniques can be divided into three categories (Sathya *et al.*, 2010):

Functional-based service selection is retrieving functional descriptions from service repositories and examining them to see if they satisfy functional requirements demanded by the client (Baldoni *et al.*, 2006; Klusch and Kapahnke, 2008).

Non-functional based (or QoS-aware) service selection is concerned with non-functional properties. With the rapidly growing number of available services, clients are presented with a choice of functionally similar services. This choice allows clients to select services that match other criteria, non-functional attributes, including QoS and context (Tian *et al.*, 2004; Yang *et al.*, 2007).

User-based service selection involves the selection of the best service among numerous discovered services based on client feedback, trust and reputation (Srivastava and Sorenson, 2010; Wang *et al.*, 2009).

8.2.2 QoS Negotiation

The requirements specified by a client may vary from the specified QoS in a service advertisement. In this case the provider and the client can enter a negotiation process to adjust the client's requirements and the provider's advertisement and reach an agreement accordingly (Swarnamugi *et al.*, 2010; Wang *et al.*, 2006b).

8.3 Related Work

So far there are no established standards for specifying NFP advertisements and requirements in a formal, machine-readable way (Bianco *et al.*, 2008). However, various XML-based languages such as Web Service Level Agreement (WSLA) (WSLA, 2003) and Web Services Offerings Language (WSOL) (Tosic *et al.*, 2002) have been proposed. These languages allow the specification of agreed-upon, non-functional properties of Web services in the form of Service Level Objectives (SLOs). They also provide a model for measuring, evaluating, and managing the compliance with non-functional properties.

A service level objective expresses a commitment to maintain a particular state of the service in a given period. The state is defined as a logical expression over predicates that refer to NFPs and defines an obligation, that is, what is asserted by the provider to the client. An example of such an obligation is (WSLA, 2003): “it is guaranteed that the average response time of the service is less than five seconds”.

Many QoS-aware service selection approaches deal with single-valued descriptions of NFPs (Wang *et al.*, 2006a; Reiff-Marganiec *et al.*, 2007). That is, a non-functional property is specified with a single value and a given unit of measurement. For example, “the price of the service is 10 dollars/month” or “the throughput of the service is at least 10MB/sec” or “the average response time of the service is at most 10 msec”. In this case, to evaluate a non-functional requirement, one performs a one-to-one comparison between the advertised and requested values. This approach works well for deterministic NFPs such as cost but not for non-deterministic ones such as response time, throughput, availability, etc.

An extension to single-valued description of NFPs is to consider ranges that indicate boundaries on NFPs. For example, “the service delay is between five and 100 msec”. In this case, different approaches could be incorporated to evaluate non-functional requirements. In (Martin-Diaz *et al.*, 2003; Kritikos and Plexousakis, 2007), the boundary values of an NFP’s range are considered for the evaluation of non-functional requirements.

There are also a few approaches which specify NFPs as intervals with certain probability characteristics. Stantchev and Schropfer (Stantchev and Schropfer, 2009) recently proposed a structure for formalization of service level objectives and technical service capabilities. A service level objective in this structure has the following form: non-functional property + predicate + metric (value, unit) + percentage + if + qualifying conditions (non-functional property + predicate + metric). An example of such a SLO would be “The transaction rate of the service is higher than 90 transactions per second in 98% of the cases if throughput is higher than 500 kB/s.”

There are a few approaches that provide distribution-based description of NFPs. Rosario *et al.*, (Rosario *et al.*, 2007) suggest that hard guarantees (e.g., response time always less than five msec) are not realistic and using soft probabilistic descriptions is more appropriate. In this work, NFPs are specified using probability distributions of the form $P(X \leq x)$, where X is a random NFP (such as response time) and x is a particular value of interest. Hwang *et al.*, (Hwang *et al.*, 2007) consider an NFP measure as a discrete random variable associated with a probability mass function (PMF) which results in discrete probability distributions. This can be applied to model NFPs such as reliability, fidelity and price. However, it is less intuitive to use a PMF for describing other NFPs, such as response time, whose domain is inherently

continuous. Li et al., (Li *et al.*, 2009) also assume a distribution-based model for description of NFPs. They propose a novel evaluation mechanism which indicates “the degree of match” between an NFP offer and a requirement, where a requirement is described using a utility function on the range of acceptable values.

Others also consider non-quantitative NFPs and introduce ontologies for semantic based matching, where semantic defines the relationship between QoS-related terms and provides a mechanism to evaluate NFP properties whose values are instances of given domain ontologies (Chaari *et al.*, 2008).

QoS service selection involves finding services that match a set of QoS requirements. Matching is a one to one comparison of an obligation and a requirement. Then, there is an aggregation process to evaluate a service with regard to all requirements. With this regard, the QoS-aware literature in SOA considers the challenges of “semantic” matching, where semantic defines the relationship between QoS-related terms (Chaari *et al.*, 2008). On the contrary, in this work we are interested in finding a better matching based on the actual distribution of non-functional properties.

Conventional service selection approaches are geared for choosing a single service to satisfy the client’s requirements. They use optimization algorithms to choose the “best” available service or rank services with regard to an application-specific utility function (Yan and Piao, 2009; Yu and Lin, 2005). Other QoS-related tracks of research in SOA use and extend the simple one-to-one matching with QoS negotiation (Swarnamugi *et al.*, 2010; Wang *et al.*, 2006b), and breaking down and dealing with global and local QoS constraints in composite services (Yang *et al.*, 2007). Marzolla and Mirandola (Marzolla, M. and Mirandola, R., 2010) recently published a survey of QoS-related literature in SOA. In this survey, they identified the main approaches

followed in the literature, including: providing QoS ontologies, developing techniques and tools for measurement and monitoring of non-functional properties of services, and devising QoS-aware service selection and composition strategies. To the best of our knowledge, there are no other approaches similar to *Request Replication* where a set of services are incorporated in order to satisfy non-functional requirements.

8.4 Proposed Request Replication Strategy

In this chapter, we present an alternative strategy for QoS-aware service selection. The proposed strategy benefits clients in potentially receiving better services for less cost. In this method, a client uses multiple functionally equivalent services to get the quality they want while minimizing the service costs. We will show how and when using multiple services can increase the quality of service.

8.4.1 Motivating Example

Assume services S_1 to S_5 are different implementations of a calendar service with usage prices of \$40, \$10, \$20, \$10, and \$10 per month, respectively. Assume that the following service advertisements are provided by corresponding providers of S_1 to S_5 .

S_1 : The response time of S_1 is less than or equal to 9s in 96% of the cases

S_2 : The response time of S_2 is less than or equal to 10s in 92% of the cases

S_3 : The response time of S_3 is less than or equal to 10s in 92% of the cases

S_4 : The response time of S_4 is less than or equal to 8s in 70% of the cases

S_5 : The response time of S_5 is less than or equal to 8s in 70% of the cases

Assume that a client is looking for a calendar service with the following QoS requirement.

R: Response time of the service must be less than or equal to 9s in 96% of the cases

With this information, a conventional service selection mechanism will choose S_1 , because it is the only service matching the QoS requirement. The price to be paid in this case is \$40 per month.

However, one can employ strategy different from this conventional service selection. One could choose any combination of services, concurrently send a request to all of them, and pick the fastest response. In general, this new strategy could improve the results. Assuming that the service response times for S_1 to S_5 are mutually independent and exponentially distributed, we can show that replicating over S_2 and S_3 provides the requested QoS for less. In this case, the resulting response time would be less than 9s in 99% of the cases and the price to be paid would be \$30 per month. The details are as follows:

First, we represent the advertisements of S_2 and S_3 as

$$P(R_2 \leq 10s) = 0.92$$

$$P(R_3 \leq 10s) = 0.92.$$

Knowing that the distributions of R_2 and R_3 are exponential, in this step, we need to find the rate parameters λ_2 and λ_3 of the corresponding distributions. We have

$$P(R_2 \leq 10s) = 0.92$$

$$1 - e^{-10\lambda_2} = 0.92 \Rightarrow \lambda_2 = \frac{-\ln 0.08}{10} \approx 0.25.$$

Similarly,

$$\lambda_3 \approx 0.25.$$

Since we pick the fastest response, the resulting response time R_{min} will be equal to the minimum of R_2 and R_3 and thus it is exponentially distributed with parameter $\lambda_2 + \lambda_3$, that is

$$P(R_{min} \leq r) = 1 - e^{-(\lambda_2 + \lambda_3)r}$$

$$P(R_{min} \leq 9) = 1 - e^{-(0.50)(9)} \approx 0.99.$$

This satisfies the client's requirement which is represented as

$$P(R_{req} \leq 9s) = 0.96.$$

Of course it is not at all clear that the exponential distribution is a reasonable choice for the individual response time distributions. However, if we have more information about the distribution of service response times we could estimate the underlying distributions. We will show in Section 8.4.4 how adding more information to current advertisements enables us to construct such estimates.

8.4.2 General Approach

In this work, we are dealing with the general problem of QoS-aware service selection, defined as:

Having functionally equivalent services S_1 to S_n , find the most cost efficient service(s) that match the QoS requirements of a client.

In other words, we need to minimize the cost of the selected services while satisfying client defined constraints on NFPs.

We assume the following format for QoS advertisements.

$$P(R_i \leq r_i) \geq p_i$$

$$E[R_i] = m_i,$$

which is read as “the probability that the response time of service i is less than or equal to r_i is greater than or equal to p_i , where the mean response time of service i is m_i ”. We also assume the following format for a non-functional requirement.

$$P(R \leq r_{req}) \geq p_{req},$$

which is read as “the probability that the response time of the selected service(s) is less than or equal to r_{req} must be greater than or equal to p_{req} ”.

8.4.3 Request Replication

The proposed strategy for QoS-aware service selection is called *Request Replication*. In this method, we choose one or more services whose aggregate QoS serves the needs of a client. We concurrently send a request to all of the selected services, pick the fastest response and cancel the other requests. A key motivation comes from the idea that taking advantage of even a small amount of additional choice for a client can lead to significant performance improvements. This idea has been explored by Mitzenmacher (Mitzenmacher, 2001), amongst others, in another context (queuing problems).

Algorithm 4 presents a pseudo code description of the proposed *Request Replication* method. Similar to conventional QoS-aware service selection, in the first step

we need to find all services with matching functional properties. We call this set the *functionally eligible services (FES)* set. We can use any functional-based service selection method to find this set. In the next step, we choose one or more services from *FES* whose aggregate QoS matches the request. This is done in two steps, as follows.

Algorithm 4 Request Replication

- 1: find all functionally eligible services and represent them as a set (*FES*).
 - 2: fit appropriate distributions to the response time data of all functionally eligible services.
 - 3: $cost = \infty$
 - 4: **for all** $fes \subseteq FES, fes \neq \emptyset$ **do**
 - 5: **if** $(cost_{sum} = \sum_{s \in fes} s.cost) \leq cost$ **then**
 - 6: compute cumulative distribution function (CDF) for the minimum response time distribution of the services in the subset fes at point r_{req} , that is $CDF_{min}(r_{req}) = 1 - \prod_{s \in fes} (1 - CDF_s(r_{req}))$, where $CDF_s(r_{req})$ is cumulative distribution function for service s evaluated at r_{req} .
 - 7: **if** $CDF_{min}(r_{req}) \geq p_{req}$ **then**
 - 8: $cost = cost_{sum}$
 - 9: $selectedServicesPool.replace(fes)$
 - 10: **end if**
 - 11: **end if**
 - 12: **end for**
-

STEP 1 - Fit appropriate distributions to the response time data of all functionally eligible services.

Current service advertisements do not indicate the actual distribution of NFPs. Therefore, we need to find estimates of the actual distributions in a way that they best describe available service advertisements. The first question here is “what sort of distribution better fits the available service response time data in SOA?”.

The choice of what distribution to fit and the method that we use to make the fit do not affect our approach - the insights provided in the rest of this section will still hold for other methods of fitting a distribution. Gorbenko et al., (Gorbenko *et al.*, 2009) provide an approach to measure the performance and dependability of Web services from the client's perspective and suggest that the Gamma distribution best describes response time data in SOA. Therefore, we represent response time data using Gamma distributions, if the actual distribution is unknown. As an illustration, we show how a Gamma distribution can be fit to a service advertisement of the form

$$P(R_i \leq r_i) \geq p_i$$

$$E[R_i] = m_i$$

The Gamma distribution is a two-parameter family of continuous probability distributions. It has a scale parameter θ and a shape parameter k . A random variable X that is Gamma-distributed with scale θ and shape k is denoted $X \sim \text{Gamma}(k, \theta)$. The mean and cumulative distribution function of the Gamma distribution can be expressed in terms of the Gamma function parameterized in terms of the shape parameter k and scale parameter θ . Both k and θ are positive values. The mean of a Gamma-distributed random variable X is $k\theta$ and the cumulative distribution function of X is

$$CDF(x) = P(X \leq x) = \frac{\gamma(k, x/\theta)}{\Gamma(k)},$$

where $\gamma(k, x/\theta)$ is the lower incomplete Gamma function, defined as

$$\gamma(s, x) = \int_0^x t^{s-1} e^{-t} dt,$$

and $\Gamma(k)$ is the Gamma function, defined as

$$\Gamma(k) = \int_0^{\infty} t^{k-1} e^{-t} dt.$$

There is no fixed way to fit a Gamma distribution to an available advertisement. In this work, we incorporate the *bisection search* method where we search for the root of the following function

$$g(k) = \frac{\gamma(k, r_i k / m_i)}{\Gamma(k)} - p_i$$

which is derived from the cumulative distribution function of the Gamma distribution (CDF) at $x = r_i$, where θ is replaced by m_i/k .

The bisection method searches for the root of $g(k)$ in an initial interval $[a, b]$ such that $g(a)$ and $g(b)$ have opposite signs. Then, it iteratively divides the interval in half in each step until it finds a sufficiently small interval that encloses the root. The method is guaranteed to converge to a root of g if g is a continuous function on the interval $[a, b]$ and $g(a)$ and $g(b)$ have opposite signs. It is not difficult to see that g is a continuous function of k .

To find the initial interval $[a, b]$, we start by setting k to integer values and computing $g(k)$. Knowing that k is a positive value, we compute $g(k = iv)$ for $iv = 1, 2, \dots$ until one of the following is true:

- $g(iv) = 0$: in this case the root has been found and is equal to iv .
- $g(iv) \cdot g(iv + 1) < 0$: in this case $a = iv$ and $b = iv + 1$.

The method now divides the interval (a, b) in two by computing the midpoint $c = (a + b)/2$ of the interval. Unless c is itself a root, there are now two possibilities: either $g(a) \cdot g(c) < 0$ in which case we select the interval (a, c) , or $g(c) \cdot g(b) < 0$ in which case we select the interval (c, b) to continue.

As an example, assume that in the advertisement above, $r_i = 100$, $p_i = 0.81$ and $m_i = 66$. In the initial step, we find out that $k = 2$ results in $CDF(100) - 0.81 = -0.00466777$ and $k = 3$ results in $CDF(100) - 0.81 = 0.02147040$. In the next steps we continue breaking this interval in half and testing $CDF(100) - 0.81$ for $k = 2.5, 2.25, \dots$. We find that $k = 2.17116$ is a very close fit. In fact, with 10^{-7} precision, the result is zero:

$$\frac{\gamma(2.17116, 217.116/66)}{\Gamma(2.17116)} - 0.81 = 0.00000003$$

STEP 2 - Check if any single service or combination of services satisfies the QoS requirements.

In this step, we need to find one or more services where the distribution of the minimum of their response times matches the requirements. We also need to select from available candidates, the set of services with minimum cost.

For this purpose, we first choose a subset of functionally equivalent services FES where the cumulative cost of services is less than the current cost, initially set to infinity. Then we compute the distribution of the minimum response times for services in the selected subset (in *Request Replication* we choose the fastest response and thus the distribution of response times for the super service is equal to the distribution of the minimum response time of its underlying services). The cumulative distribution function for the minimum response time for a set of services fes is computed as

$$\begin{aligned} CDF_{min}(R_{min} = r) &= P(R_{min} \leq r) \\ &= P(\text{Min}_{s \in fes}(R_s) \leq r) \\ &= 1 - P(\wedge_{s \in fes}(R_s > r)), \end{aligned}$$

where fes is a subset of functionally eligible services and R_s is the response time of service s .

It is in general difficult to calculate this value unless the response times of services in fes are mutually independent. In this case:

$$\begin{aligned} 1 - P(\wedge_{s \in fes} (R_s > r)) &= 1 - \prod_{s \in fes} P(R_s > r) \\ &= 1 - \prod_{s \in fes} (1 - CDF_s(R_s = r)). \end{aligned}$$

At this point we compute $CDF_{min}(r_{req})$ for all eligible subsets and update the pool of selected services, if a subset satisfies the requirement (i.e., $CDF_{min}(r_{req}) \geq p_{req}$).

8.4.4 Motivating Example Revisited

Adding to the previous example in Section 8.4.1, assume that we also know the means of the distributions:

$$m_1 = 8, m_2 = 8, m_3 = 8, m_4 = 7.2, m_5 = 7.2.$$

Following Algorithm 4, in the first step we find matching Gamma distributions for all service advertisements. In this case:

$$k_1 \approx 0.01, \theta_1 \approx 800$$

$$k_2 \approx 34, \theta_2 \approx 0.24$$

$$k_3 \approx 34, \theta_3 \approx 0.24$$

$$k_4 \approx 18, \theta_4 \approx 0.4$$

$$k_5 \approx 18, \theta_5 \approx 0.4$$

In the next step we try different subsets of services, starting from single services, the results are:

S_1 is still a match.

S_2 or S_3 does not match ($CDF_2(9) \approx 0.78$ and $CDF_3(9) \approx 0.78$).

S_{2+3} (replication over S_2 and S_3) does not match ($CDF_{min}(9) \approx 0.95$).

S_4 or S_5 does not match ($CDF_4(9) \approx 0.86$ and $CDF_5(9) \approx 0.86$).

S_{4+5} (replication over S_4 and S_5) matches ($CDF_{min}(9) \approx 0.98$).

In this case, replication over S_4 and S_5 is preferred since it provides the required QoS with a lower price of \$20. Note that the previous choice of S_2 and S_3 for replication is no longer an option. The extra information provided by the means results in the construction of more accurate distributions for S_2 and S_3 which consequently indicates that replication over S_2 and S_3 does not actually satisfy the requirement.

8.4.5 Discussion

In this section we discuss a number of points related to the *Request Replication* approach.

QoS Heterogeneity

One of the advantages of our algorithm is that, no matter what format the advertisements have and how one estimates a distribution for an advertisement, one is still

able to use the *Request Replication* algorithm and the underlying mathematics are valid. In other words, our proposed algorithm can deal with a mix of advertisement formats that providers may use in an SOA-based environment. For example, assume service S_1 is advertised with having average response time of 10 msec and service S_2 with mean of 9 msec and variance of 12 msec. In this case, we can fit an exponential distribution to S_1 and a Gamma distribution to S_2 and the remainder of the calculations in Algorithm 4 are still valid. The proposed *Request Replication* method also works for other distribution-based NFPs such as availability, reliability, etc. Note that the minimum distribution here must be changed to the appropriate metric. As future work, we intend to investigate how *Request Replication* can be enhanced with optimization techniques to handle requests with multiple QoS constraints.

Service Interdependencies

The underlying mathematics works well if service response times are mutually independent. In other words, for any two services S_i and S_j , $P(\text{Min}(R_i, R_j) > r) = P(R_i > r)P(R_j > r)$ if R_i and R_j are independent. If response times are dependent, the calculation is not as straightforward. In this case, we would also need to estimate joint probabilities, which may be difficult to calculate. Note that our approach should also work well if there is approximate independence, i.e., if the relation above approximately holds. As future work, we are aiming to extend the current approach by modeling services' interdependencies.

High Variance

The *Request Replication* method takes advantage of high variability in the response time data. If the response times were deterministic, the result of choosing more than one service would be the same as the result of choosing the service with minimum response time (which is what conventional QoS-aware service selection does). High variability in response times increases the chance that multiple services may complement each other with respect to performance and thus achieving better response times via *Request Replication* is more likely. Therefore, the provided method improves the results over conventional service selection as long as the response time data are highly variable. The precise form of the distribution is not important.

Replication Overheads

Similar to other domains such as fault tolerance (Looker *et al.*, 2005; Salatge and Fabre, 2007; Zheng and Lyu, 2009), replication in this work also introduces a number of overheads.

- Using *Request Replication*, one should think of a mediator which replicates a service call to selected services, returns the first response to the client and cancels the remaining calls.
- Replication increases incoming network traffic to servers. However, we incorporate request cancellation to avoid lengthy responses and reduce congestion as much as possible.

As suggested by (Looker *et al.*, 2005) the replication overhead is small and thus acceptable.

Request Cancellation Model

We use request cancellation as a companion to our request replication strategy. From the client's point of view, canceling a synchronous (request-response) SOAP call is the same as for any other HTTP call. The client just disconnects and stops listening for the response. A well written server will check whether the client is still connected before proceeding with lengthy operations (e.g., in .NET the server would check `IsClientConnected`) and should cancel the operation if not. One-way calls in asynchronous calls cannot be canceled in this manner however, because the client has already sent the payload and disconnected. Cancellation of one-way calls would require an explicit call to some sort of cancellation method on the SOAP service (such as `Abort`), which it would have to explicitly support. Note that our strategy provides the best results when we have the "plan" pricing model for Web service usage as opposed to the "pay per usage" model. For the latter we need to take into account the cancellation fees when computing $cost_{sum}$ in Algorithm 4.

Stateful Services

Our approach handles stateless services. Compared with stateful services, stateless services offer higher scalability and availability, are faster and easier to test, and can deal better with fail-over situations. Therefore, stateless services are preferred when developing Web services. Also, a stateful service may be implemented as a stateless service by sending all relevant information with every request.

8.5 Revisiting Service Advertisements

Looking at the literature, current QoS advertisements typically provide a single value (e.g., the average) to represent the response time of a service. For example, “Average response time of operation X from service Y is less than or equal to 0.5 seconds” or “Response time of operation X from service Y is less than or equal to 0.5 seconds in at least 95% of the cases”. This makes it difficult for clients to employ a strategy other than the conventional service selection where the client is limited to choosing “a particular service” as the “best available choice” with regard to their needs.

Conventional service selection would work well if the non-functional properties of services were actually deterministic. However, the literature (Gorbenko *et al.*, 2009; Zhu *et al.*, 2006) suggests that non-functional properties of services and in particular, service response time, are non-deterministic and highly variable due to dynamics of the environment in which services are deployed. For example, factors such as network traffic and server congestion greatly influence the response time of services and current advertisements do not reflect this high variance.

Knowing more about the actual distribution of NFPs provides more opportunities for clients, including:

- **Adjusting the requirements:** non-functional requirements are usually soft constraints. Clients may be willing to change their non-functional requirements based on the availabilities and the offered prices, as in the case of QoS negotiation. Providing more information about the actual distribution of NFPs makes it easier for the clients to make better decisions about their requirements. As an example, assume the following situation. Services S_1 and S_2 are advertised with average response times of 0.95 and 1.05 seconds, respectively. A potential

client needs a service with an average response time of less than one second. With this information, the client would choose S_1 as the better choice. On the other hand, assume that we know variances of the response times for services S_1 and S_2 . In this case, S_1 has high variance ($\sigma = 1$) and S_2 has low variance ($\sigma = 0$). Knowing this information, the client may be willing to revise their requirement and choose S_2 as the better choice as it is more reliable with regard to timing constraints.

- **Changing the selection strategy:** as mentioned before, having a better understanding of the distribution of non-functional values makes it possible for clients to choose from a variety of strategies for QoS-aware service selection. *Request Replication* is one example of such strategies which benefits clients by providing them with better quality services with less cost. There may also be other possibilities.

For these reasons we recommend that providers advertise the response time of their services using more than one representative value. This makes it possible to estimate a distribution for the response time of a service. For example, with two pieces of information, one can estimate a Gamma distribution, which is shown to be a good fit for describing the response time distribution in SOA (Gorbenko *et al.*, 2009). Providers can use any two pieces of information about the response time. In this work we are considering advertisements of the form

$$P(R \leq limit) \geq percentage$$

$$E[R] = m,$$

as this is a lightweight change to the advertisements currently available. However, one could think of other pieces of information, such as mean m and variance σ^2 , for an advertisement.

No matter how the services are advertised, the *Request Replication* strategy can still be applied. As long as we estimate a distribution to the available advertisement we can use Algorithm 4 to find appropriate services for replication. If an advertisement provides one piece of information about the response time, we could fit an exponential distribution (a simple form of Gamma distribution, where $k = 1$), and if two pieces of information are provided a fit to a Gamma distribution could be performed. Our algorithm is able to deal with a mix of advertisement formats, which is very likely in a heterogeneous SOA environment.

8.6 Conclusion

In this chapter, we presented a novel alternative strategy to satisfy the QoS requirements of a client in a more cost efficient manner by taking advantage of the existing high variance in the values of non-functional properties. In this method, we use multiple independent low-cost, low-quality services to satisfy the QoS requirements of a single request.

There is a possibility to combine the *Request Replication* idea with the QoS negotiation process already existing in the SOA literature. In this approach one can negotiate with the provider of a service to acquire a reasonably low price for a service (and of course loose QoS as a result) and then incorporate multiple services of this sort to acquire the required QoS. This idea is being investigated as future work.

Also, the underlying assumption of independence should be considered more comprehensively. If service response times are correlated as a result of services sharing resources (e.g., shared services, network access, server, etc), the calculation of the minimum response time distribution becomes more difficult. As future work we are investigating how we can exploit knowledge about structure of the correlations to perform the required calculations.

Moreover, one can think of more complex QoS scenarios where the requirements involve multiple non-functional constraints that must be satisfied at the same time. Many researchers have used optimization approaches such as linear programming (Cardellini *et al.*, 2007) and constraint satisfaction (Zemni *et al.*, 2010) techniques to deal with such scenarios in the traditional QoS service selection literature. *Request Replication* simply suggests more options (the use of multiple services in addition to single ones) when selecting services and can be an appropriate replacement for traditional QoS-aware selection in optimization-based approaches. This idea is being investigated as future work.

Bibliography

- Abreu, R., Gonzalez, A., Zoetewij, P., and van Gemund, A. J. C. (2008). Automatic software fault localization using generic program invariants. In *ACM Symposium on Applied Computing (SAC)*, pages 712–717.
- Abreu, R., Zoetewij, P., Golsteijn, R., and van Gemund, A. J. C. (2009a). A practical evaluation of spectrum-based fault localization. *Systems and Software*, **82**(11), 1780–1792.
- Abreu, R., Zoetewij, P., and van Gemund, A. J. C. (2009b). Spectrum-based multiple fault localization. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 88–99.
- Aggarwal, C. C. and Wang, H., editors (2010). *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer, 1st edition.
- Agrawal, H. (1992). *Towards Automatic Debugging of Computer Programs*. Ph.D. thesis, Purdue University, West Lafayette, IN.
- Agrawal, H., DeMillo, R. A., and Spafford, E. H. (1993a). Debugging with dynamic slicing and backtracking. *Software - Practice & Experience*, **23**(6), 589–616.

- Agrawal, H., Horgan, J. R., London, S., and Wong, W. E. (1995). Fault localization using execution slices and data flow tests. In *6th International Symposium on Software Reliability Engineering (ISSRE)*, pages 143–151.
- Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *International Conference on Very Large Data Bases*, pages 487–499.
- Agrawal, R., Imielinski, T., and Swami, A. (1993b). Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 207–216.
- Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, **18**, 333–340.
- Allen, F. E. (1970). Control flow analysis. *ACM SIGPLAN Notices*, **5**(7), 1–19.
- Allen, F. E. (1974). Interprocedural data flow analysis. In *IFIP Congress*, pages 398–402.
- Antoniol, G. and Gueheneuc, Y.-G. (2005). Feature identification: A novel approach and a case study. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 357–366.
- Antoniol, G. and Gueheneuc, Y. G. (2006). Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, **32**(9), 627–641.
- ARMiner (n.d.). ARMiner project. Retrieved December 10, 2012, from <http://www.cs.umb.edu/laur/ARMiner/>.

- ARtool (n.d.). ARtool project. Retrieved December 10, 2012, from <http://www.cs.umb.edu/laur/ARtool/>.
- Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., and Arikawa, S. (2002). Efficient substructure discovery from large semi-structured data. In *2nd SIAM International Conference on Data Mining (SDM)*, pages 158–174.
- Asai, T., Arimura, H., Uno, T., and Nakano, S.-I. (2003). Discovering frequent substructures in large unordered trees. In *6th International Conference on Discovery Science (DS)*, pages 47–61.
- Baah, G. K., Podgurski, A., and Harrold, M. J. (2008). The probabilistic program dependence graph and its application to fault diagnosis. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 189–200.
- Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley.
- Baldoni, M., Baroglio, C., Martelli, A., and Patti, V. (2006). Reasoning about interaction protocols for customizing web service selection and composition. *Logic and Algebraic Programming*, **70**(1), 53–73.
- Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold Co., 2nd edition.
- Bianco, P., Lewis, G. A., and Merson, P. (2008). Service level agreements in service-oriented architecture environments. Technical Report CMU/SEI-2008-TN-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

- Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E. (1994). The concept assignment problem in program understanding. In *15th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 482–498.
- Binkley, D. (2007). Source code analysis: A road map. In *Future of Software Engineering (FOSE)*, pages 104–119.
- Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Machine Learning Research*, **3**, 993–1022.
- Cardellini, V., Casalicchio, E., Grassi, V., and Presti, F. L. (2007). Scalable service selection for web service composition supporting differentiated QoS classes. Technical Report RR-07.59, Universita di Roma.
- Chaari, S., Badr, Y., and Biennier, F. (2008). Enhancing web service selection by QoS-based ontology and WS-Policy. In *ACM Symposium on Applied Computing (SAC)*, pages 2426–2431.
- Chen, K. and Rajlich, V. (2000). Case study of feature location using dependence graph. In *8th IEEE International Workshop on Program Comprehension (IWPC)*, pages 241–249.
- Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks (DSN)*, pages 595–604.
- Cheng, H., Lo, D., Zhou, Y., Wang, X., and Yan, X. (2009). Identifying bug signatures using discriminative graph mining. In *18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 141–152.

- Cheng, X. (2010). *Exploring Hybrid Dynamic and Static Techniques for Software Verification*. Ph.D. thesis, Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA.
- Chi, Y., Yang, Y., and Muntz, R. R. (2003). Indexing and mining free trees. In *3rd IEEE International Conference on Data Mining (ICDM)*, pages 509–512.
- Chi, Y., Muntz, R., Nijssen, S., and Kok, J. (2004a). Frequent subtree mining - an overview. *Fundamenta Informaticae - Advances in Mining Graphs, Trees and Sequences*, **66**(1-2), 161–198.
- Chi, Y., Yang, Y., and Muntz, R. R. (2004b). Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In *16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 11–20.
- Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *27th International Conference on Software Engineering*, pages 342–351.
- Cole, O. (2009). Aprobe: A framework for non-intrusive software instrumentation. Retrieved November 23, 2012, from <http://www.ocsystems.com/>.
- ConceptExplorer (n.d.). Formal concept analysis toolkit version 1.0.1. Retrieved November 23, 2012, from <http://sourceforge.net/projects/conexp>.
- Cook, D. J. and Holder, L. B., editors (2006). *Mining Graph Data*. John Wiley & Sons.
- Cornelissen, B. (2009). *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Ph.D. thesis, Delft University of Technology, Delft, Netherlands.

- Cornett, S. (2010). Code coverage analysis. Retrieved November 23, 2012, from <http://www.bullseye.com/coverage.html>.
- Crochemore, M. (1981). An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, **12**(5), 244–250.
- Dallmeier, V., Lindig, C., and Zeller, A. (2005). Lightweight defect localization for java. In *19th European conference on Object-Oriented Programming (ECOOP)*, pages 528–550.
- Darwin, I. F. (2004). *Java Cookbook*. O’Reilly & Associates.
- Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *American Society for Information Science*, **41**, 391–407.
- Di Fatta, G., Leue, S., and Stegantova, E. (2006). Discriminative pattern mining in software fault detection. In *3rd International Workshop on Software Quality Assurance (SOQUA)*, pages 62–69.
- Dietz, L., Dallmeier, V., Zeller, A., and Scheffer, T. (2009). Localizing bugs in program executions with graphical models. In *23rd Conference on Neural Information Processing Systems (NIPS)*, pages 468–476.
- Dit, B., Reville, M., Gethers, M., and Poshyvanyk, D. (2011). Feature location in source code: A taxonomy and survey. *Software Maintenance and Evolution: Research and Practice (JSME)*. doi: 10.1002/smr.567.
- Do, H., Elbaum, S., and Rothermel, G. (2004). Infrastructure support for controlled

- experimentation with software testing and regression testing techniques. In *International Symposium on Empirical Software Engineering*, pages 60–70.
- Dong, G. and Pei, J. (2007). *Sequence Data Mining*, volume 33 of *Advances in Database Systems*. Springer.
- Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Wiley & Sons.
- Dudar, M. (2012). Feature in software product - definition. Retrieved November 23, 2012, from <http://www.pmsnack.com/feature-in-software-product/>.
- Eaddy, M. (2008). *An Empirical Assessment of the Crosscutting Concern Problem*. Ph.D. thesis, Columbia University, New York City, NY.
- Eaddy, M., Aho, A. V., Antoniol, G., and Gueheneuc, Y.-G. (2008). Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *16th IEEE International Conference on Program Comprehension (ICPC)*, pages 53–62.
- Edwards, D., Simmons, S., and Wilde, N. (2006). An approach to feature location in distributed systems. *Systems and Software*, **79**(1), 57–68.
- Eichinger, F. (2011). *Data-Mining Techniques for Call-Graph-Based Software-Defect Localisation*. Ph.D. thesis, Karlsruhe Institute of Technology (KIT).
- Eichinger, F., Bohm, K., and Huber, M. (2008). Mining edge-weighted call graphs to localise software bugs. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, pages 333–348.

- Eichinger, F., Pankratius, V., Grobe, P. W. L., and Bohm, K. (2010a). Localizing defects in multithreaded programs by mining dynamic call graphs. In *Testing: Academic & Industrial Conference and Practice and Research Techniques (TAIC PART)*, pages 56–71.
- Eichinger, F., Krogmann, K., Klug, R., and Bohm, K. (2010b). Software-defect localization by mining dataflow-enabled call graphs. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, pages 425–441.
- Eichinger, F., Obner, C., and Bohm, K. (2011). Scalable software-defect localization by hierarchical mining of dynamic call graphs. In *SIAM Conference on Data Mining (SDM)*, pages 723–734.
- Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, **29**(3), 210–224.
- Eisenberg, A. D. and De Volder, K. (2005). Dynamic feature traces: Finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 337–346.
- Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Language Systems*, **9**(3), 319–349.
- FIXImulator (n.d.). Fiximulator. Retrieved November 23, 2012, from <http://fiximulator.org/>.

- FIXMessages (n.d.). Fiximates, fix interactive message and tag explorer. Retrieved November 23, 2012, from <http://fixprotocol.org/FIXimate3.0/>.
- Flach, P. A. and Lachiche, N. (2001). Confirmation-guided discovery of first-order rules with tertius. *Machine Learning*, **42**(1-2), 61–95.
- Gorbenko, A., Kharchenko, V., Mamutov, S., Tarasyuk, O., Chen, Y., and Romanovsky, A. (2009). Real distribution of response time instability in service-oriented architecture. Technical Report CS-TR-1182, Newcastle University, Newcastle upon Tyne, England.
- Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A call graph execution profiler. In *SIGPLAN symposium on Compiler construction (SIGPLAN)*, pages 120–126.
- Gupta, N., He, H., Zhang, X., and Gupta, R. (2005). Locating faulty code using failure-inducing chops. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 263–272.
- Hamrouni, T., Yahia, S. B., and Nguifo, E. M. (2008). Garm: Generalized association rule mining. In *6th International Conference on Concept Lattices and their Applications (CLA)*, pages 145–156.
- Hamrouni, T., Yahia, S. B., and Nguifo, E. M. (2010). Generalization of association rules through disjunction. *Annals of Mathematics And Artificial Intelligence*, **59**(2), 201–222.
- Hong, T.-P., Kuo, C.-S., and Chi, S.-C. (1999). Mining association rules from quantitative data. *Intelligent Data Analysis*, **3**(5), 363–376.

- Howden, W. E. (1978). A survey of dynamic analysis methods. In E. Miller and W. E. Howden, editors, *Software Testing and Validation Techniques*, pages 184–206. IEEE Computer Society Press.
- Hsu, H.-Y., Jones, A., and Orso, A. (2008). Rapid: Identifying bug signatures to support debugging activities. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 439–442.
- Hulme, G. V. (2002). Software’s challenge. *Information Week*. Retrieved November 9, 2012, from <http://www.informationweek.com/software-challenge/6500499>.
- Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *19th International Conference on Software Engineering (ICSE)*, pages 191–200.
- Hwang, S. Y., Wang, H., Tang, J., and Srivastava, J. (2007). A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. *Journal of Information Science*, **177**(23), 5484–5503.
- Jensen, F. V. (2009). Bayesian networks. *Interdisciplinary Reviews: Computational Statistics*, **1**(3), 307–315.
- Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM Conference on Automated Software Engineering*, pages 273–282.
- Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *24th International Conference on Software Engineering (ICSE)*, pages 467–477.

- Jordan, M. I., editor (1999). *Learning in Graphical Models*. MIT Press.
- Junit (n.d.). Junit. Retrieved November 23, 2012, from <http://www.junit.org/>.
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–353.
- Klusch, M. and Kapahnke, P. (2008). Semantic web service selection with SAWSDL-MX. In *International Semantic Web Conference*, page 316.
- Knuth, D. E., Morris, J. J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, **6**(2), 323–350.
- Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, **29**(3), 155–163.
- Krcal, L. (2011). *Tree Edit Distance and Approximate Tree Pattern Matching Problem*. Bachelor’s thesis. Department of Computer Science and Engineering, Czech Technical University in Prague, Prague, Czech Republic.
- Kritikos, K. and Plexousakis, D. (2007). Semantic QoS-based web service discovery algorithms. In *5th IEEE European Conference on Web Services (ECOWS)*, pages 181–190.
- Li, P., Comerio, M., Maurino, A., and Paoli, F. D. (2009). Advanced non-functional property evaluation of web services. In *7th IEEE European Conference on Web Services*, pages 27–36.

- Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26.
- Liu, C., Yan, X., Yu, H., Han, J., and Yu, P. S. (2005). Mining behavior graphs for “backtrace” of noncrashing bugs. In *5th SIAM International Conference on Data Mining (SDM)*.
- Liu, C., Fei, L., Yan, X., Han, J., and Midkiff, S. P. (2006). Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, **32**(10), 831–848.
- Livshits, B. and Zimmermann, T. (2005). Dynamine: Finding common error patterns by mining software revision histories. In *10th European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 296–305.
- Lo, D., Cheng, H., Han, J., Khoo, S.-C., and Sun, C. (2009). Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 557–566.
- Looker, N., Munro, M., and Xu, J. (2005). Increasing web service dependability through consensus voting. In *29th Annual International Conference on Computer Software and Applications Conference (COMPSAC)*, pages 66–69.
- Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *13th International*

- Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339.
- Lu, S. Y. (1979). A tree-to-tree distance and its application to cluster analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **1**, 219–224.
- Luccio, F., Enriquez, A. M., Rieumont, P. O., and Pagli, L. (2001). Exact rooted subtree matching in sublinear time. Technical Report TR-01-14, Universita Di Pisa.
- Luccio, F., Enriquez, A. M., Rieumont, P. O., and Pagli, L. (2004). Bottom-up subtree isomorphism for unordered labeled trees. Technical Report TR-04-13, Universita Di Pisa.
- Lucia, B. and Ceze, L. (2009). Finding concurrency bugs with context-aware communication graphs. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 553–563.
- Lucia, B., Devietti, J., Strauss, K., and Ceze, L. (2008). Atom-aid: Detecting and surviving atomicity violations. In *35th Annual International Symposium on Computer Architecture (ISCA)*, pages 277–288.
- Lyle, J. R. and Weiser, M. (1987). Automatic program bug location by program slicing. In *2nd International Conference on Computer and Applications*, pages 877–883.
- Martin-Diaz, O., Cortes, A. R., Benavides, D., Duran, A., and Toro, M. (2003). A quality-aware approach to web services procurement. In *4th Intl. VWB Workshop Technologies for E-Services (TES)*, pages 42–53.

- Marzolla, M. and Mirandola, R. (2010). QoS analysis for web service applications: a survey of performance-oriented approaches from an architectural viewpoint. Technical Report UBLCS-2010-05, University of Bologna, Bologna, Italy.
- Mehregani, N. (2006). Eclipse test and performance tools platform project. Retrieved November 23, 2012, from <http://www.eclipse.org/tptp/platform/documents/probekit/probekit.html>.
- Miller, F. P., Vandome, A. F., and McBrewster, J. (2009). *Levenshtein Distance: Information theory, Computer science, String (computer science), String metric, Damerau-Levenshtein distance, Spell checker, Hamming distance*. Alpha Press.
- Mitzenmacher, M. (2001). The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, **12**(10), 1094–1104.
- Naish, L., Lee, H. J., and Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Transaction on Software Engineering and methodology (TOSEM)*, **20**(3), 1–32. Article No. 11.
- Nanavati, A. A., Chitrapura, K. P., Joshi, S., and Krishnapuram, R. (2001). Mining generalised disjunctive association rules. In *10th International Conference on Information and Knowledge Management (CIKM)*, pages 482–489.
- NanoXML (n.d.). Nanoxml. Retrieved November 23, 2012, from <http://sir.unl.edu/portal/bios/nanoxml.php>.
- Nijssen, S. and Kok, J. N. (2003). Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences (MGTS)*, pages 55–64.

- Nijssen, S. and Kok, J. N. (2004). A quickstart in frequent structure mining can make a difference. In *10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 647–652.
- Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. *SIGSOFT Software Engineering Notes*, **9**(3), 177–184.
- Palix, N., Lawall, J., and Muller, G. (2010). Tracking code patterns over multiple software versions with herodotos. In *9th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 169–180.
- Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, **33**(6), 420–432.
- Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc.
- Rajlich, V. and Wilde, N. (2002). The role of concepts in program comprehension. In *IEEE International Workshop on Program Comprehension (IWPC)*, pages 271–278.
- Reiff-Marganiec, S., Yu, H. Q., and Tilly, M. (2007). Service selection based on non-functional properties. In *Proceedings of Non Functional Properties and Service Level Agreements in Service Oriented Computing Workshop*, pages 128–138.

- Renieris, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39.
- Reps, T., Ball, T., Das, M., and Larus, J. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *6th European Software Engineering Conference*, pages 432–449.
- Rhino (n.d.). Mozilla rhino. Retrieved November 23, 2012, from <https://developer.mozilla.org/en-US/docs/Rhino>.
- Rielly, M. (2011). Combining dynamic testing and static verification. Retrieved November 23, 2012, from <http://www.vectorcast.com/>.
- Robillard, M. P. (2008). Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **17**(4), 1–36.
- Rosario, S., Benveniste, A., Haar, S., and Jard, C. (2007). Probabilistic QoS and soft contracts for transaction based web services. In *IEEE International Conference on Web Services (ICWS)*, pages 126–133.
- Rutar, N., Almazan, C. B., and Foster, J. S. (2004). A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256.
- Safyallah, H. and Sartipi, K. (2006). Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension (ICPC)*, pages 84–88.

- Saha, D., Nanda, M. G., Dhoolia, P., Nandivada, V. K., and Chandra, V. S. S. (2011). Fault localization for data-centric programs. In *19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 157–167.
- Salatge, N. and Fabre, J. (2007). Fault tolerance connectors for unreliable web services. In *International Conference on Dependable Systems and Networks (DSN)*, pages 51–60.
- Salton, G. and McGill, M. (1986). *Introduction to Modern Information Retrieval*. McGraw-Hill.
- Sampaio, M. C., Cardoso, F. H. B., dos Santos Jr., G. P., and Hattori, L. (2008). Mining disjunctive association rules. Technical report, Faculty of Informatics, University of Lugano, Lugano, Switzerland.
- Santelices, R. A., Jones, J. A., Yu, Y., and Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types. In *31st International Conference on Software Engineering (ICSE)*, pages 56–66.
- Sartipi, K. and Safyallah, H. (2010). Dynamic knowledge extraction from software systems using sequential pattern mining. *Software Engineering and Knowledge Engineering (IJSEKE)*, **20**(6), 761–782.
- Sathya, M., Swarnamugi, M., Dhavachelvan, P., and Sureshkumar, G. (2010). Evaluation of QoS based web service selection techniques for service composition. *International Journal of Software Engineering (IJSE)*, **1**(5), 73–90.

- Shasha, D., Wang, J. T.-L., Zhang, K., and Shih, F. Y. (1994). Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics*, **24**(4), 668–678.
- Sommerville, I. (2010). *Software Engineering*. Addison Wesley, 9th edition.
- Srivastava, A. and Sorenson, P. G. (2010). Service selection based on customer rating of quality of service attributes. In *IEEE International Conference on Web Services (ICWS)*, pages 1–8.
- Stallman, R. M. and the GCC Developer Community (2003). Using the gnu compiler collection. Retrieved November 23, 2012, from <http://gcc.gnu.org/>.
- Stantchev, V. and Schropfer, C. (2009). Negotiating and enforcing QoS and SLAs in grid and cloud computing. In *4th International Conference on Advances in Grid and Pervasive Computing*, pages 25–35.
- Sterling, C. D. and Olsson, R. A. (2005). Automated bug isolation via program chipping. In *6th International Symposium on Automated Analysis-Driven Debugging*, pages 23–32.
- Swarnamugi, M., Sathya, M., and Dhavachelvan, P. (2010). A negotiation model for web service selection. In *International Conference on Recent Trends in Soft Computing and Information Technology*, pages 251–256.
- Tian, M., Gramm, A., Ritter, H., and Schiller, J. H. (2004). Efficient selection and monitoring of QoS-aware web services with the WS-QoS framework. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 152–158.

- Toivonen, H. (1996). *Discovery of Frequent Patterns in Large Data Collections*. Ph.D. thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland.
- Tornado (n.d.). Tornado web server. Retrieved November 23, 2012, from <http://www.tornadoweb.org/>.
- Tosic, V., Patel, K., and Pagurek, B. (2002). WSOL web service offerings language. In *Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web (CAiSE/WES)*, pages 56–67.
- Ujhazi, B., Ferenc, R., Poshyvanyk, D., and Gyimothy, T. (2010). New conceptual coupling and cohesion metrics for object-oriented systems. In *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 33–42.
- Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag.
- Wang, P., Chao, K. M., Lo, C. C., Farmer, R., and Kuo, P. T. (2009). A reputation-based service selection scheme. In *IEEE International Conference on e-Business Engineering (ICEBE)*, pages 501–506.
- Wang, T. and Roychoudhury, A. (2005). Automated path generation for software fault localization. In *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 347–351.
- Wang, X., Vitvar, T., Kerrigan, M., and Toma, I. (2006a). A QoS-aware selection model for semantic web services. In *International Conference on Service Oriented Computing (ICSOC)*, pages 390–401.

- Wang, Y., Wang, L., and HuA, C. (2006b). QoS negotiation protocol for grid workflow. In *5th International Conference on Grid and Cooperative Computing (GCC)*, pages 195–198.
- Washio, T., Kok, J. N., and Raedt, L. D. (2005). *Advances in Mining Graphs, Trees And Sequences*. IOS Press.
- WebLech (n.d.). Weblech url spider. Retrieved November 23, 2012, from <http://weblech.sourceforge.net/>.
- Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, **25**(7), 446–452.
- Weka (n.d.). Weka machine learning suite. Retrieved November 23, 2012, from <http://www.cs.waikato.ac.nz/ml/weka/>.
- Wilde, N. and Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, **7**(1), 49–62.
- Wilde, N., Gomez, J. A., and Gus, T. (1992). Locating user functionality in old code. In *IEEE International Conference on Software Maintenance*, pages 200–205.
- Wilde, N., Simmons, S., Pressel, M., and Vandeville, J. (2008). Understanding features in SOA: Some experiences from distributed systems. In *International Workshop on Systems Development in SOA*, pages 59–62.
- Wong, W. E. and Debroy, V. (2010). Software fault localization. In P. A. Laplante, editor, *Encyclopedia of Software Engineering*, volume 1, pages 1147–1156. Auerbach Publications.

- Wong, W. E. and Qi, Y. (2006). Effective program debugging based on execution slices and inter-block data dependency. *Systems and Software*, **79**(7), 891–903.
- Wong, W. E., Wei, T., Qi, Y., and Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. In *First International Conference on Software Testing, Verification and Validation*, pages 42–51.
- Wong, W. E., Debroy, V., and Choi, B. (2010). A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, **83**(2), 188–208.
- Wong, W. E., Debroy, V., Li, Y., and Gao, R. (2012a). Software fault localization using DStar (D*). In *6th IEEE International Conference on Software Security and Reliability (SERE)*, pages 21–30.
- Wong, W. E., Debroy, V., and Xu, D. (2012b). Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications & Reviews*, **42**(3), 378–396.
- WSLA (2003). Web service level agreement (WSLA) language specification. Technical Report wsla-2003/01/28, IBM Corporation.
- Xie, T. and Notkin, D. (2002). An empirical study of java dynamic call graph extractors. Technical Report UW-CSE-02-12-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA.
- Yan, J. and Piao, J. (2009). Towards QoS-based web services discovery. *Lecture Notes in Computer Science*, **5472**, 200–210.

- Yan, X., Cheng, H., Han, J., and Yu, P. S. (2008). Mining significant graph patterns by leap search. In *ACM SIGMOD International Conference on Management of Data*, pages 433–444.
- Yang, Y., Tang, S., Xu, Y., Zhang, W., and Fang, L. (2007). An approach to QoS-aware service selection in dynamic web service composition. In *3rd IEEE International Conference on Networking and Services (ICNS)*, pages 18–23.
- You, K., Qian, Z., Tang, B., Lu, S., and Chen, D. (2009). QoS-aware replication in service composition. *International Journal of Software and Informatics*, **3**(4), 465–482.
- Yousefi, A. and Down, D. G. (2011a). Request replication: An alternative to QoS aware service selection. In *IEEE International Conference on Service Oriented Computing and Applications (SOCA)*, pages 1–4.
- Yousefi, A. and Down, D. G. (2011b). Request replication: An alternative to QoS aware service selection. Technical report, McMaster University, Hamilton, ON, Canada. CAS-11-07-DD.
- Yousefi, A. and Sartipi, K. (2011). Identifying distributed features in soa by mining dynamic call trees. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 73–82.
- Yousefi, A. and Wassyng, A. (2013). A call graph mining and matching based defect localization technique. In *6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 86–95.

- Yu, K., Lin, M., Gao, Q., Zhang, H., and Zhang, X. (2011). Locating faults using multiple spectra-specific models. In *ACM Symposium on Applied Computing (SAC)*, pages 1404–1410.
- Yu, T. and Lin, K. J. (2005). Service selection algorithms for composing complex services with multiple QoS constraints. In *3rd International Conference on Service Oriented Computing*, pages 130–143.
- Zaki, M. J., Ramakrishnan, N., and Zhao, L. (2010). Mining frequent boolean expressions: Application to gene expression and regulatory modeling. *International Journal on Knowledge Discovery in Bioinformatics*, **1**(3), 68–96.
- Zeller, A. (1999). Yesterday, my program worked. today, it does not. why? In *7th European Software Engineering Conference and the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–267.
- Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10.
- Zeller, A. (2009). *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2nd edition.
- Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, **28**(2), 183–200.
- Zemni, M. A., Benbernou, S., and Carro, M. (2010). A soft constraint-based approach

- to QoS-aware service selection. In *International Conference on Service Oriented Computing (ICSOC)*, pages 596–602.
- Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault location. In *6th International Symposium on Automated Analysis-Driven Debugging*, pages 33–42.
- Zheng, Z. and Lyu, M. (2009). A QoS-aware fault tolerant middleware for dependable service composition. In *International Conference on Dependable Systems and Networks (DSN)*, pages 239–248.
- Zhu, L., Gorton, I., Liu, Y., and Bui, N. B. (2006). Model driven benchmark generation for web services. In *International Workshop on Service Oriented Software Engineering (SOSE)*, pages 33–39.

Appendix A

Risk Analysis

This appendix presents the known risks associated with the proposed defect localization technique, possible consequences of those risks and suggestions for mitigating them. The following subsections are organized according to the steps of our defect localization technique and present risks in each step.

A.1 Tracing

A.1.1 Instrumenting Target System

Risks. Not every method is instrumented.

Consequences. i) The defective method is filtered out. ii) Method calls differentiating between the failing and correct executions are filtered out (e.g., method b is expected to be called following the defective method a , but is missing because of a defect in a . In this case, if b is filtered, one cannot differentiate between the correct and failing executions based on their call graphs which both consist of a call to a).

Mitigation. i) Instrument all methods: it mitigates both consequences but can result in very long tracing times and very large traces that require special care to be useful for defect localization. ii) Instrument one subsystem at a time, then aggregate the results: it can mitigate both consequences. More investigation is required. iii) Filter out methods with low probability of being defective. This includes (a) simple methods, that are small, do not transform data, do not interact with remote components, and are not time dependent, e.g., small utility methods, and (b) methods that have been tested and are proved correct, e.g., library methods. This mitigation deals with the first consequence. iv) Identify and filter irrelevant subsystems, packages, classes, or methods, based on the debugger's knowledge about the target system and the nature of failure.

A.1.2 Running Test Cases on Instrumented System to Get Execution Traces

Risks. i) Tracing takes too much time. ii) Traces are too large to handle.

Consequences. If the traces are too large, their associated call trees can take up available run time memory. This can prevent the tool from further processing.

Mitigation. i) Do not instrument all methods: it mitigates both risks but causes new risks specified in A.1.1. ii) Break down the traces into a number of sub-traces: it mitigates the second risk. More investigation is required.

A.2 Constructing and Reducing Dynamic Call Trees

A.2.1 Constructing Dynamic Call Trees

Risks. Call trees do not fit in memory.

Consequences. If the trees are too big, they can take up available run time memory. This can prevent the tool from further processing.

Mitigation. i) Do not instrument all methods: it mitigates the risk but causes new risks specified in A.1.1. 2) Break down the call trees into a number of subtrees. More investigation is required.

A.2.2 Reducing Dynamic Call Trees (Iteration Reduction)

Risks. Repetitive method calls which differentiate between the failing and correct executions are removed from the trees.

Consequences. We cannot handle frequency-affecting defects. This applies to both methods that are called from within a loop and those that are repeated a few times outside a loop.

Mitigation. i) Set a threshold for the maximum number of repetitive calls allowed (for example five), so that methods that are repeated less than this threshold are not deleted from the trees. This mitigation targets repetitive method calls outside a loop. ii) Assign the frequency of method calls to tree edges and extend the proposed analysis to consider frequency-affecting defects.

A.3 Mining Frequent Subtrees

Risks. i) Since the *minimum height* threshold is equal to two in our method, frequent subtrees with a single method call are not recorded as patterns. ii) Since the *minimum frequency* threshold is equal to two in our method, subtrees that are observed in only one tree are not considered frequent and thus not recorded. iii) Subtrees that are part of bigger trees with the same support set are not recorded. iv) Since sufficient test cases are not available, some correct patterns do not reveal themselves. v) We do not consider failing patterns, i.e., subtrees that only appear in failing test cases.

Consequences. Some correct patterns may be missing. If the missing patterns are related to features other than the target (failing) feature or are specific to target feature but are sub-patterns of other patterns in our database, this will not cause any problems. If the missing patterns are related to the target feature, but other patterns with the same root exist in our database, we report the root as defect related. If the missing patterns are related to the target feature and no other patterns could localize the defect, the defect cannot be localized.

Mitigation. To mitigate risk *i*, set the *minimum height* threshold to zero. This can lead to the generation of more patterns that can increase the number of false positives. To mitigate risk *ii*, apply a test case twice with different inputs. This can lead to the creation of big patterns that are the result of a serialization of features. To mitigate risks *iii* and *iv*, if patterns with the same root as the missing pattern exist and are high in the ranking, we report the root method as defect related (as we will see in A.5.2, such patterns could be low in the ranking). To mitigate risks *iii*, *iv* and *v*, consider methods that are not part of any correct patterns but are observed in the failing trace, especially those that, according to relevance metrics, are highly

relevant to the target feature. To mitigate risk *iv*, ensure decision coverage in the test suite.

A.4 Pattern Analysis

Risks. i) Due to improper ranking or assuming a very small number for *rootsToPick*, some feature-specific patterns are missing. ii) Due to improper ranking or assuming a very big number for *rootsToPick*, some shared/irrelevant patterns are selected as feature-specific.

Consequences. Consequence of risk *i* is that the defective method may get filtered out. Consequence of risk *ii* is that we may increase the number of false positives.

Mitigation. Develop a feedback and/or learning mechanism to improve the results of ranking and cutting the ranked list.

A.5 Defect Localization

A.5.1 Pattern Matching

Risks. i) Defect does not change the call tree. ii) Defect does not change the call tree in the section associated with the failing feature (i.e., defect is related to and changes the code before this section). iii) Defect is related to a subtask that is shared between multiple features. iv) The correct pattern is not in the database of mined patterns.

Consequences. Consequence of risks *i*, *ii* and *iii* is that all feature-specific patterns match and thus the best guesses we can provide for the location of the defect would

be roots of all feature-specific patterns. For risk *iv*, if other patterns with the same root exist we can identify the root as defect-related; otherwise, we cannot localize the defect.

Mitigation. Risk *i* implies that the defect is not structure-affecting and thus this is out of the scope of this work. To mitigate risk *ii*, consider shared patterns and feature-specific patterns for features that have been executed before the faulty feature. To mitigate risk *iii*, consider shared patterns. Run extra tests to identify if other features are affected by the defect and consider the patterns specific to the group of features involved. To mitigate risk *iv*, if patterns with the same root exist and are high in the ranked list, we report the root; otherwise, consider methods that are not part of any correct patterns but are executed in the failing trace, especially those that are highly relevant to the target feature.

A.5.2 Analysis of Matching Results

Risks. i) Due to improper ranking, the pattern-match pair which includes the defect-related method is not ranked high enough in the list of suspicious matchings. ii) Not all the differences between an expected pattern and the failing execution are related to the defect.

Consequences. Too many false positives are possible. Therefore, defect-related method is not in the top methods reported.

Mitigation. To mitigate risk *i*, a better ranking algorithm should be devised, which considers other ranking criteria besides *parentDiffSize* and *diffSize*. To mitigate risk *ii*, we can (a) keep a history of similar failures (i.e., failures that relate to the same feature and produce similar outcomes) and search for frequent failing patterns, or (b)

consider ignoring changes related to sub-patterns that are common to all executions.

A.6 General Risks

Risks. i) The frequency of the appearance of features in test scenarios is not considered when analyzing pattern-feature relevance. ii) The number of times a pattern appears in an execution is not considered when analyzing pattern-feature relevance. iii) Failures due to feature interaction, including those related to ordering of features or multiple execution of the same feature in a scenario are not considered.

Consequences. This affects feature location and can lead to improper ranking of patterns.

Mitigation. See the above discussion for A.4.

Appendix B

Pattern Analysis using Association Rules Mining

In Section 5.5, we explained how we use method-feature relevance formulations to quantify the relevance between features of a system and frequent patterns discovered on dynamic call trees. In this Appendix, we explain how a complex association rules mining based algorithm could help making more definite decisions about patterns implementing a feature and why we avoided such an approach.

B.1 Main Idea

The idea here is to discover pattern-feature relations in terms of association rules. After performing pattern mining on dynamic call trees and identifying the test-feature and test-pattern maps presented in Chapter 5, we can build a *test-feature-pattern* table, where each row represents a test case and each column represents a feature or a pattern. Table B.1 illustrates an example. A check mark in cell c_{ij} indicates that

test case t_i includes feature/pattern f_j/p_j .

Table B.1: Example of a test-pattern-feature table

Test Cases	Features			Patterns			
	f1	f2	f3	p1	p2	p3	p4
t1	✓		✓	✓	✓	✓	
t2	✓	✓		✓	✓	✓	
t3		✓	✓		✓		
t4	✓	✓		✓	✓	✓	✓
t5			✓	✓			
t6	✓						✓

In the next step we perform association rules mining on this data. An association rule can be of the form $p \Leftrightarrow f$, which means pattern p is observed if and only if feature f is executed. Such a pattern is a feature-specific candidate. Reviewing the pattern types identified in Section 5.5, we can re-define different pattern-feature relations in terms of association rules, as follows:

- A feature-specific candidate pattern (a pattern p that exists only in traces associated with feature f) can be specified as $p \Leftrightarrow f$.
- A common pattern (a pattern p that is observed as part of the execution of a number of features, identified as set F') can be specified as $\bigvee_{f \in F' \subseteq F} f \Leftrightarrow p$.
- An omni-present pattern (a pattern p that exist in all traces) can be specified as $\bigvee_{f \in F} f \Leftrightarrow p$.

Extending this, A feature can be represented as a formula on patterns. For example, $f \equiv p1 \wedge (p2 \oplus p3)$ means executing feature f results in the observation of pattern $p1$ and either pattern $p2$ or pattern $p3$. This suggests that if feature f fails, one of the patterns $p1$ and $p2$ or $p3$ should be investigated to locate the problem. This is

an alternative approach for finding relevant patterns for defect localization which is different from the relevance formulation based approach we presented in Section 5.5. In this approach we are ideally seeking rules of the form $\psi \Rightarrow \omega$ and $\psi \Leftrightarrow \omega$, where ψ and ω are boolean expressions over the union of features and patterns, which are built using conjunctive (\wedge) and disjunctive (\vee and \oplus) operators, and have no literals in common. An example of such a rule is, $f1 \wedge p2 \Leftrightarrow p3 \otimes p4$, which means if $f1$ is executed and $p2$ is observed, we are also expecting $p3$ or $p4$. Such information can enhance the effectiveness of defect localization. If $f1$, $p2$ and $p3$ are observed in a faulty execution, there is no need to look for $p4$ even though it may be feature-specific. This reduces the time that we spend on defect localization and also the number of false positives. *Generalized association rules mining*, introduced in the next section, helps discover such rules from our *test-feature-pattern* table. We are interested in discovering a complete set of rules, where no two rules are logically equivalent.

B.2 Generalized Association Rules Mining

Association rules mining is a popular data mining method seeking interesting relations between variables in a large database. Agrawal et al., (Agrawal *et al.*, 1993b) introduced association rules for discovering regularities between product sales in point-of-sales. A rule such as *milk, eggs* \Rightarrow *toast* found in the transactions database of a supermarket would indicate that a customer that buys milk and egg is likely to also buy toast bread.

Toivonen (Toivonen, 1996) generalized the notion of set and association rules to boolean formula and boolean rules. A boolean rule is an expression of the form $\psi \Rightarrow \omega$, where ψ and ω are Boolean formulae. A Boolean formula can be built using

negation, conjunctive and disjunctive operators.

The following are among researchers who deal with generalized association rules mining. Zaki et al., (Zaki *et al.*, 2010) introduce a novel framework, called BLO-SOM, for mining frequent Boolean expressions over binary valued databases, where boolean expressions can be conjunctions, disjunctions, conjunction of disjunctions and disjunction of conjunctions. Flach and Lachiche (Flach and Lachiche, 2001) deal with the discovery of first-order logic rules from data. The proposed system, Tertius, employs a normal form variant of the first-order rules which is comprised of a disjunction of possibly negated literals, e.g., $H_1 \vee H_2 \vee \neg B_1 \vee \neg B_2$, usually written as $B_1 \wedge B_2 \Rightarrow H_1 \vee H_2$. Nanavati et al., (Nanavati *et al.*, 2001) deals with generalized disjunctive association rules (d-rules) which allow the disjunctive logical operators, \vee (inclusive-or) and \oplus (exclusive-or). Hamrouni et al., (Hamrouni *et al.*, 2008, 2010) proposed a tool called GARM to discover association rules offering conjunctive, disjunctive and negative connectors between items.

B.3 Discussion

Available techniques and tools for mining association rules deal with a limited set of rule structures. The majority of tools (ConceptExplorer, n.d.; ARtool, n.d.; ARMiner, n.d.) discover the simplest form of relation between attributes, which is the conventional association rule mining of the form $p \Rightarrow q$ where, p and q are built using the conjunctive operator \wedge only. Others who consider generalized association rules pose limitations on the structure of a rule. For example, GARM (Hamrouni *et al.*, 2010) discovers rules of the form $p \Rightarrow q$ in three cases: 1) both p and q are built using disjunction operator \vee ; 2) both p and q are built using negation operator \neg ; and

3) either p or q is built using \vee operator and the other one uses \neg . Nanavati et al., (Nanavati *et al.*, 2001) discover q for known p in $p \Rightarrow q$, where q is in disjunctive normal form (DNF). Weka (Weka, n.d.), a widely known data mining tool suite, implements a number of different association rules mining algorithms. Weka considers both the simple conjunctive association rules discussed above and rules of the form $p \Rightarrow q$ where, p is built using conjunctive operator \wedge and q uses disjunction \vee .

None of the above tools and techniques produces a complete set of boolean association rules. This is because this problem is exponential in nature (Sampaio *et al.*, 2008). For example, extending Nanavati's approach requires examining 2^{2^n} combinations of attributes where n is the number of attributes (in our case the number of features plus the number of patterns). This is not practical. For this reason, one has no other choice but to limit the types of rules one can mine. Also, for many structures such as DNF, one cannot guarantee completeness of the set of rules (Nanavati *et al.*, 2001). Discovering rules of the form $p \Rightarrow q$ where, p is an exclusive disjunction (XOR) of patterns and q is a conjunction of features, which is one of the most interesting relations for our analysis is itself exponential in nature. For the mentioned complexity reasons we decided to change our strategy to using method-feature relevance formulae from the domain of feature location, and rank patterns according to the relevance of their methods to features as explained in Section 5.5.

Appendix C

Dealing with Concurrency

In Chapter 5, we explained a graph-based approach to localize software defects. In this approach, we focused on defects in single-threaded applications. In this Appendix, we examine how we can extend our graph-based technique to deal with concurrency defects such as atomicity and ordering violations.

C.1 Concurrency Defects

Several types of concurrency defects exist in the literature. As specified by Lucia and Ceze (Lucia and Ceze, 2009), the main categories discussed in the literature are data races, atomicity violations and ordering violations. A data race happens when two or more threads access a shared memory location without using suitable locks or other synchronization to control their accesses to that memory. Atomicity violation is referred to the state when a group of memory operations that are assumed to be executed atomically are interleaved by memory accesses from other threads. This happens when atomic operations are not enclosed inside the same critical section.

Ordering violations occur when memory accesses in different threads happen in an unexpected order. A study on real world concurrency defects (Lu *et al.*, 2008) shows that, most of existing tools do not address less well-studied classes of defects such as ordering violations and defects involving multiple variables fairly well.

C.2 Select Related Work in Localizing Concurrency Defects

Lucia *et al.*, (Lucia *et al.*, 2008) present Atom-Aid, which is an approach to detect potential atomicity violations and pro-actively choose chunk boundaries before the potential violations get exposed. A chunk is a set of operations executed on hardware as if they are one unbreakable operation. It provides implicit atomicity. To detect potential atomicity violations Atom-Aid performs the following. With every read/write it saves the addresses in r_c/w_c after copying their current values to r_p/w_p (does this for every thread). Then it checks if these values indicate any unserializable interleavings (e.g., if a remote write interleaves two local reads, a remote write interleaves a local write followed by a local read, a remote write interleaves a local read followed by a local write, or a remote read interleaves two local reads). This approach deals with single-variable atomicity violations.

Lucia and Ceze (Lucia and Ceze, 2009) use communication graphs to detect incorrect thread interleavings. A communication graph indicates thread communication through shared memory operations (i.e., reads and writes). It is a directed acyclic bipartite graph where nodes indicate memory read/write instructions and edges indicate communication via shared memory. Edge (u, v) indicates that v reads from a

memory location that u has most recently wrote to. The basic idea in (Lucia and Ceze, 2009) is to indicate communication edges that appear in incorrect executions only. Such edges indicate incorrect interleavings and thus where a concurrency defect is originated. They argue that context-aware variations of communication graphs are able to detect more complicated concurrency defects such as multi-variable atomicity and order violations as opposed to basic context-oblivious graphs.

C.3 The Proposed Idea

Concurrency defects (e.g., single and multi variable atomicity and order violations, data races, etc) may or may not affect the call tree. The real reason behind a concurrency related failure is usually a wrong thread interleaving. The proposed technique for defect localization in this thesis is based on changes in dynamic call trees of different executions. This approach as it stands is not suitable for locating concurrency defects. However, a similar idea (graph mining and matching) can be applied to deal with concurrency defects.

The basic idea here is to extend dynamic call trees with communication edges. In this notation a conventional dynamic call tree is augmented with edges that represent communication via shared memory. To localize concurrency defects, we mine patterns of communication from correct executions and compare them against the communications that have happened in the faulty run. This is an extension of Lucia and Ceze (Lucia and Ceze, 2009). When using communication graphs to detect concurrency defects, Lucia and Ceze (Lucia and Ceze, 2009) suggest taking one of a couple of alternatives: 1) consider edges that are rare in communication graphs, believing that buggy communication is rare; 2) produce a set of defect-only graphs by

taking a graph difference between the graph of each buggy execution and the union of all graphs obtained from non-buggy executions.

As with call graph affecting defects (Chapter 5), we believe that we can provide better results when applying feature location to find related subgraphs for comparison. Hence, we suggest augmenting dynamic call trees with communication edges and mining two types of patterns:

- Intra-thread patterns: patterns on the conventional dynamic call trees which assist in locating call tree and frequency affecting defects.
- Cross-thread patterns: patterns on communication edges which indicate correct structure between the timing of events (i.e., memory read and writes) in a multi-threaded application. The cross-thread patterns identify order and atomicity invariants and any violations from those invariants can be detected using pattern matching.

This idea requires further investigation.