Parallel Paths Analysis Using Function Call Graphs

by

Arman Naeimian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

© Arman Naeimian 2019

Author's Declaration

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

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

Abstract

Call graphs have been used widely in different software engineering areas. Since call graphs provide us with detailed information about the structure of software elements and components and how they are connected with each other, they could be used in detecting specific structures and patterns in the code such as malware, code clones, unreachable code, and many other software symptoms that could be searched by their structural features. In this work, we have analyzed parallel paths in function call graphs in three Java open-source projects. Parallel paths emerge when there is more than one path between two nodes in the call graph. We investigated the reasons such paths are created and used for and also the problems that result in removing them. Moreover, we have used the results of our analyses to find instances of parallel paths in the projects that we analyzed and suggest some changes to developers based on that. Based on our results, we found three categories of problems associated with parallel paths and four categories of usages of them.

Acknowledgements

I definitely would not have finished this thesis had it not been for the people who supported and helped me. I will try to thank everybody that comes to my mind here.

Foremost, I would like to thank my supervisors, Mei Nagappan and Semih Salihoglu, for guiding me through this process while also providing me with every facility that I needed to conduct this research.

To Mike Godfrey and Grant Weddell, thank you for being readers for my thesis and giving me valuable feedback.

Thank you to my parents, Hedieh and Jahangir for all of your support and sympathy.

To all of my labmates, Ten, Aaron, Reza, Davood, Dan, Bushra, Gema, Cassiano, Cosmos, Ashwin, Magnus, Sunjay, Jeremy, Kilby, Achyudh, and Lakshman, thank you for making the office an enjoyable place for me.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Thesis Contributions . . . 3

1.2 Thesis Organization . . . 3

2 Background and Related Research 4

2.1 Call Graphs and Malware Detection . . . 4

2.2 Call Graphs Visualization . . . 5

2.3 Call Graphs and Software Metrics . . . 6

2.4 Network Motifs and Software Motifs . . . 7

2.5 Call Graphs and Detection of Bugs and Patterns . . . 8

3 Methodology 11

3.1 Corpus and Data . . . 11

3.2 Building Projects . . . 13

3.3 Generating Call Graphs . . . 13

3.4 Parsing Call Graphs . . . 16

3.5 Calculating Diffs . . . 16

4 Results 18
4.1 RQ1: Why are parallel paths created and used? 19
4.2 RQ2: Why are parallel paths removed or changed? 24
4.2.1 Hadoop 25
4.2.2 Flink 28
4.2.3 Jmeter 31
4.2.4 Conclusions 32
4.3 Comparison of the first and the last versions 35
4.3.1 Hadoop 35
4.3.2 Flink 37
4.3.3 Jmeter 38
4.3.4 Conclusions 39
4.4 RQ3: Is it possible to suggest to developers to apply some changes to the current version of the software based on the information gained from analyzed parallel paths? 40
5 Conclusions 43
5.1 Threats to Validity 44
5.2 Future Work 44
References 45
APPENDICES 47

List of Tables

3.1 Dataset projects 12

4.1 Total number of triangles and analyzed triangles for each project 19
4.2 Categorization of triangles 19
4.3 Results of automated analysis for each category 22
4.4 Hadoop Removed Triangles 26
4.5 Hadoop Removed Triangles Categories 27
4.6 Commit type distribution between Hadoop categories 29
4.7 Flink Removed Triangles 32
4.8 Flink Removed Triangles Categories 32
4.9 Commit type distribution between Flink categories 32
4.10 Jmeter Removed Triangles 33
4.11 Jmeter Removed Triangles Categories 33
4.12 Commit type distribution between Jmeter categories 33
4.13 Hadoop first and last versions comparison 35
4.14 Hadoop removed triangles of first version categories 36
4.15 Flink first and last versions comparison 37
4.16 Flink removed triangles of first version categories 37
4.17 Jmeter first and last versions comparison 38
4.18 Jmeter removed triangles of first version categories 38
4.19 Double Check Group Instances 41
4.20 Hook Methods Group Instances 41

List of Figures

1.1 An example of parallel paths 2

2.1 An example of functions visualization in different levels of density 6

2.2 Software motifs structure 8

3.1 The process of generating call graphs 12

3.2 A sample code snippet and its corresponding call graph 14

3.3 An example of a change in code which changes the call graph 15

3.4 An example of a change in code which does not change the call graph . . . 15

3.5 The process of finding changed patterns 17

4.1 The simplest form of parallel paths 18

4.2 Sample code showing SubSuperClass category 21

4.3 Sample code showing Overloaded Methods category 22

4.4 Sample code showing Delegation Category 23

4.5 Parallel Paths Changes 25

4.6 Extract Method 26

4.7 Double Check Category 28

4.8 Hook Method Category 29

4.9 Logging Issues Category. In function ***a***, warn is replaced with debug. . . . 30

4.10 Parameters Category. In function ***a***, the call to ***c*** should be removed because it is called with a deprecated parameter. 31

Chapter 1

Introduction

Using function call graphs for software analysis has gained considerable attention in the research community. By definition, a call graph is a control flow graph, which represents calling relationships between subroutines in a computer program. Each node represents a procedure, and each edge (f, g) indicates that procedure f calls procedure g. [27]. In function call graphs (FCGs), nodes are functions and edges are calls between functions.

Since call graphs provide us with symbolic, syntactic, and topological features of software systems [28], they could be used in different areas. In many works, call graphs have been used for malware detection [11][10][31][16]. Furthermore, some efforts have focused on defining new software metrics based on call graphs to measure software complexity [24][21][22]. Also, some other researchers have focused on using call graphs for clone detection, similarity detection, and unreachable code detection [23][26].

Nevertheless, call graphs have been mainly used for analyzing software systems in high levels to gain a big picture of the system and understand how its coarse-grained elements (e.g. modules and components) interact [29] [8] [10]. There are still many paths for using call graphs at lower levels, like in the level of functions. In this work we have used call graphs in the level of single functions to investigate purposes and issues related to parallel paths. Parallel paths emerge when there are more than one path between two nodes (functions) in the function call graph. Figure 1.1 shows an example of parallel paths. In this example, there are two paths between nodes ***main*** and ***printf*** , one with length 1 (***main-printf***) and one with length 3 (***main-parse-execute-printf***).

Therefore, the question would be why should one function call another function in two ways. We are curious to know why parallel paths are created and used to find whether there are reasonable and correct ways of using them. In addition, we want to know why

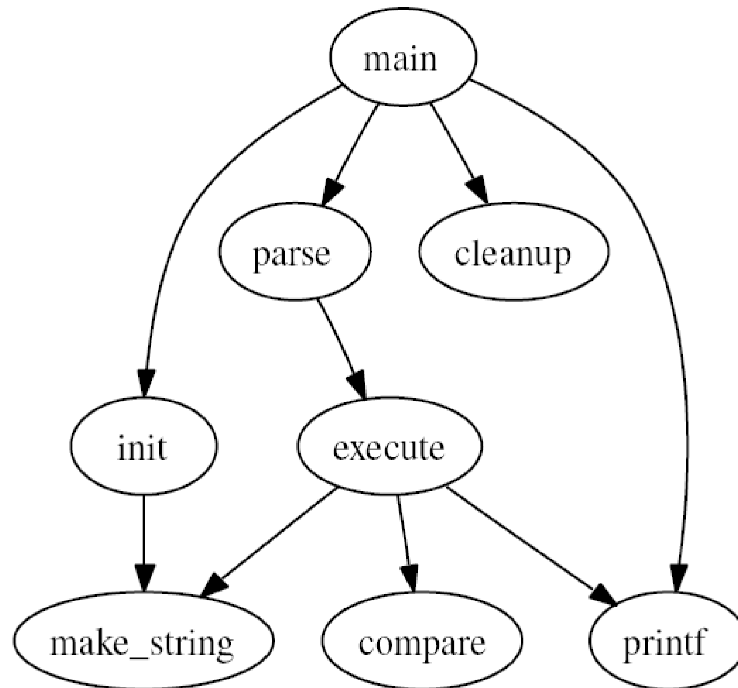


Figure 1.1: An example of parallel paths

parallel paths are removed after a while from being added. Moreover, we want to know whether the changes or removals applied to parallel paths could be used to suggest some changes to developers. In this regard, we explore three research questions:

- **RQ1** *Why are parallel paths created and used?*
- **RQ2** *Why are parallel paths removed or changed?*
- **RQ3** *Is it possible to suggest to developers to apply some changes to the current version of the software based on the information gained from analyzed parallel paths?*

1.1 Thesis Contributions

This thesis has two main contributions:

- We propose an approach for analyzing parallel paths in function call graphs and conduct an empirical study on three large Java projects using this approach to investigate uses of parallel paths and their changes and removals.
- Based on our results, we suggest some changes to developers of analyzed projects to improve the quality of the code, make it more readable and understandable, or remove some bugs or issues from that.

1.2 Thesis Organization

This thesis is organized into five sections. In section 2, we talk about the background and related research. In section 3, we elaborate on the methodology that we have used for generating and using call graphs, the corpus and projects that we have used, and the analysis approach that we have selected for each research question. In section 4, we present the results for each research question and discuss them. Section 5 includes conclusions of this research, threats to validity of results, and future work and some suggestions for extending this work.

Chapter 2

Background and Related Research

Investigating call graphs and their uses and features has a long history in software engineering research [9][13][18] . However, there are some recent works which have tried to use call graphs to help software engineers in different phases of software development such as debugging, maintenance, and code comprehension [8][25][26].

2.1 Call Graphs and Malware Detection

Gascon et al. propose a malware detection method for Android applications using function call graphs. They use machine learning classification techniques to improve the efficiency of finding similarities between call graphs which is used for finding similarities between code samples [11].

DU et al.[10] also present a new malware detection method which solves the problems and limitations of existing algorithms such as computation time and the need for manual operation. In addition, their method outperforms other approaches in terms of malware detection accuracy. They divide a function call graph into community structures and then use the features of these structures to detect malware.

Zhou et al. based on the fact that reuse is widely adopted in the creation of malware and they usually have similar structures, have used an approach to compare between an existing known malware and a suspect malware in order to detect that. They propose a framework called CGIDroid, which categorizes malware into families using API call graph isomorphism and comparing characteristics of suspect malware with the samples of each family [31].

Kinable and Kostakis use call graph clustering in order to classify malware into similar categories. They represent malware as call graphs which enables them to detect structurally similar malware and put them in the same category. This categorization helps anti-viruses to detect different releases of malware which are deliberately created by authors to bypass anti-viruses detection mechanisms [16].

The difference between our work and the works in this category is that we go through the level of functions and investigate the interactions between them. However, these works usually focus on higher levels (e.g., classes and modules) and the relations between them.

2.2 Call Graphs Visualization

Bhattacharya et al. use graph-based analysis to capture software projects evolution and facilitate the process of development. They generate call graphs to show the project structure in levels product (e.g., the source code), and process (e.g., developer collaboration) and define some graph metrics based on these call graphs to capture their properties. Their results enable them to detect structural changes and estimate bug severity, prioritize debugging and refactoring efforts, and predict defect-prone releases [8].

Code2graph is a prototype python tool which automates the process of analyzing Python source code and its structure, generating static call graphs and visualizing them, and constructing a similarity matrix of all possible execution paths in the system [12].

Shah and Guyer [25] have proposed an interactive call-graph visualization tool for viewing large programs written in Java or C++. Their tool can scale to view huge numbers of functions and their connectivity. Their technique is inspired by a DNA microarray visualization from biology, which is a grid of pixels that packs information into a single display densely. They use this grid for visualizing functions in a program. Figure 2.1 shows an example of such a grid. The far-left visualization shows the initial program. The user selected a number of cells and then generated the middle visualization. The far-right visualization is the result of filtering out more cells. They also keep metadata for each node, which includes the number of callers, callees, and if the function makes a call to itself. The interactive features provide the user with information about each function. For example, hovering mouse over a cell highlights it in yellow. A details pane shows the metadata associated with each function as we hover over each node. Right-clicking the cell shows the function name and left-clicking the mouse highlights the cell in green, and clicking again deselects the cell. When we mouse over a cell, the callees will be highlighted with a black edge between them. Callers of the method will be drawn with a red edge between them to distinguish the callees by holding down a keyboard shortcut.

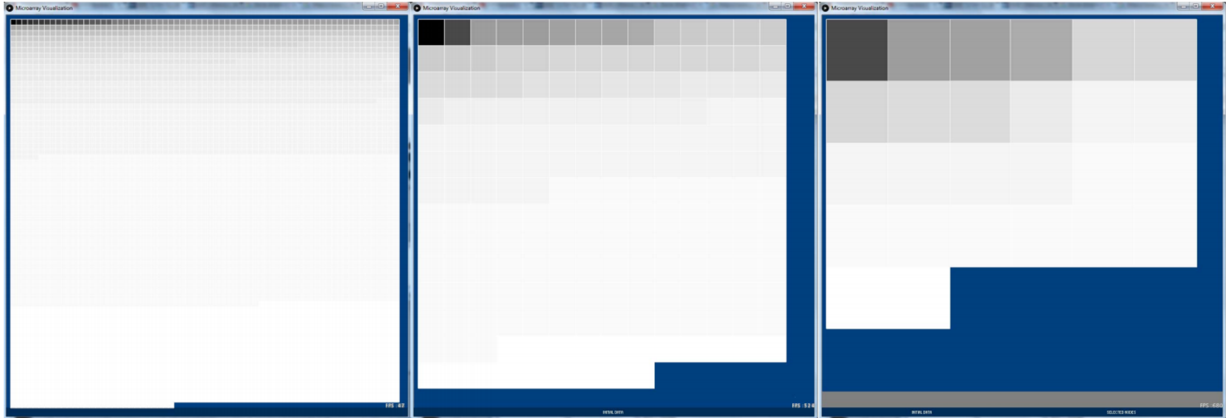


Figure 2.1: An example of functions visualization in different levels of density

Hejderup et al. use call graphs for dependency management between open-source software libraries. They mention that dependency graphs, which denote libraries as nodes and dependencies between them as edges, are insufficient for assessing the severity, impact and spread of bugs. Hence, they propose a fine-grained dependency network which includes call graph information within and across dependencies and this granularity in source code level enables them to investigate the effects and the propagation of bugs more accurately [15].

Our focus on analyzing parallel paths in call graphs is the difference between our work and the papers in this category, which focus on the visualization of call graphs.

2.3 Call Graphs and Software Metrics

Qu et al. [22] propose two new class cohesion metrics, MCC (Method Community Cohesion) and MCEC (Method Community Entropy Cohesion) based on community structures of software call graphs. By conducting an empirical study followed by two case studies, they show that these metrics outperform existing commonly used metrics. First, they provide additional information about class cohesion which existing metrics can not. Moreover, they lead to better results in class fault prediction.

Qingfeng et al. [21] have proposed an approach to defining three new software metrics to reflect the complexity of software from the level of method call relationship. They generate the call graph of the system and analyze defined metrics by call graph.

Sawadpong et al. [24] have used exception handling call graphs to propose some new exception-based software metrics. Exception handling call graph is a graph in which functions containing at least one form of Java exception handling constructs (e.g., try-catch) are represented with black nodes and other functions are denoted with white nodes. Pruning the methods without any exception handling construct results in a set of disjoint subgraphs. They use these graphs to define metrics like sSize (Number of nodes in exception handling subgraph), sComplexity (Number of edges of exception handling subgraph), mSize (Number of nodes within a class in exception handling subgraph), and mComplexity(Number of edges within a class). Their results reveal that these metrics can predict the fault-proneness of exception classes better than conventional software metrics.

There are also many works focused on software complexity metrics. Madhan et al. [17] have used Halstead metrics to analyze the complexity of some genetic optimization algorithms. Hariprasad et al. [14] have used Halstead metrics to compare two versions of a program in terms of unpredictability, execution time, and exertion.

We do not focus on metrics and defining them, but our focus is on parallel paths uses, changes, and issues. We categorize parallel paths in different groups, but we do not define new metrics based on them, which is the difference between our approach and the works in this category.

2.4 Network Motifs and Software Motifs

Zhang and Xuelin. [30] have used the concept of software networks and network motifs to analyze the distribution of bugs in software systems. They define a software network as a graph which contains software entities such as classes, functions, and variables as nodes and the relationships between them as edges. Also, they define network motifs as recurrent and statistically significant sub-graphs or patterns or sub-graphs which continuously repeat themselves in specific networks or even among various networks. They choose a specific type of network motifs named FeedForward Loop (FFL) motif which is one type of loop motifs with the highest degree of uniqueness and is discovered and turned out to be statistically significant in various types of complex networks. The structure of this motif is the same as the structure of the simplest form of parallel paths that we are analyzing in our work. Their results show that FFL motifs have a significant correlation with software bugs. It can build a connection between software code structure and software logic.

Wu et al. [28] also use software motifs and function call graphs to detect software homology which determines whether a pair of software evolves from the same code, belongs

to the same family, or is originated from the same author or organization. Their work is based on the fact that call graphs provide us with both symbolic and syntactic information of the code (e.g., writing styles, layout characteristics and author's preferences). They first generate the call graph from source code or binary executable files. Then, they extract software motifs from the call graph. They only focus on motifs with three vertices, similar to what we do in this work. Figure 2.2 shows three vertices motifs structures that they have used. They repeat this process for both plaintiff and defendant (two software pieces that they want to calculate their similarity).

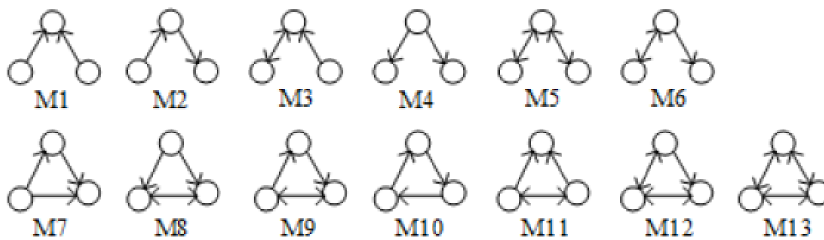


Figure 2.2: Software motifs structure

They also calculate a metric called *motif frequency distribution*, which is the probability distribution of the frequencies of different motif types in a graph. Using this metric, they define a homology score which is used for determining the homology of two software pieces. They show that their approach is efficient to detect software homology, especially for large-scale source codes. Their results reveal that average homology detection accuracy of 10 well-known larger-scale homology source codes with 470 versions is 98.47

The works in this group are close to what we have done because software motifs are different combinations of nodes and edges, and we focus on one of them, which is the simplest form of parallel paths. The difference is that our focus is on parallel paths, their uses, and the problems related to them, but to the best of our knowledge, none of the mentioned works focus on this topic.

2.5 Call Graphs and Detection of Bugs and Patterns

Oruc et al. [20] have used a graph mining approach for detecting design patterns in object-oriented code. Their process starts with analyzing source code and extracting Abstract Syntax Trees out of it and then, based on ASTs, they create a graph model. After that, they define templates for all Gang of Four design patterns by analyzing class and sequence

diagrams of all 23 GoF patterns. Finally, they use Subdues sgiso [7] sub-graph mining algorithm to search for pattern templates in a model graph. They have developed a fully automated tool named DesPaD (Design Pattern Detector) to do this process. In their model graph, they use classes, abstract classes, template classes, and interfaces as nodes and the relations between classes (e.g., inheritance, aggregation, association, and composition) as edges. They have used JUnit 3.8, JUnit 4.1, and AWT 3.1 as their data sample. Their tool can detect all GoF design patterns, and their results show that it outperforms other similar tools by creating 47% better recall values.

Romano et al. [23] have defined a static approach named DUM (Detecting Unreachable Methods) which detects unreachable methods by traversing the software call graph. This tool works at Java byte-code level and takes internal and external classes as input. Internal classes are in the source code of a project and external classes are related to external packages and libraries. They consider almost all kinds of method calls (e.g., Virtual, Static, Constructor) and also native methods which do not have a body in the graph. They consider three types of nodes as reachable nodes. First, the main methods (specified in MANIFEST files). Second, methods invoked to initialize a field and methods invoked in initializer blocks. Third, methods used to customize the serialization/deserialization process of objects (e.g., writeObject(java.io.ObjectOutputStream)) that are then invoked by reflection. Every method that has a direct link from a reachable node is considered reachable and whatever else is considered as unreachable. They performed the analysis of finding unreachable methods on four open-source projects and then compared their results with JTombstone and CodePro, which are other tools for detecting unreachable methods. The results show that DUM outperforms other tools in terms of correctness, completeness, and accuracy of results.

Turhan et al. [26] have used call graphs and data mining techniques to locate software bugs. In order to create method level defect predictors to predict defect proneness of modules, they have used static call graph based ranking (CGBR) and nearest neighbor sampling. They have performed their analyzes on 25 large telecommunication systems, and their results show that at least 70% of the defects can be detected by inspecting only 6% of the code using a Naive Bayes model and 3% of the code using CGBR framework.

Musco et al. [19] have used call graphs to propose an evaluation technique to predict impact propagation. Impact analysis predicts the software elements (e.g., modules, classes, methods) that are impacted by a change in the software. They have defined four types of call graphs for propagation analysis. In the first type, overriding methods are not considered and included in the graph. In the second type, the call graph uses the signature of the class according to the static type of the receiver. The third type of call graphs considers classes hierarchy, inheritance and interfaces implementation. The fourth type, in

addition to classes hierarchy, considers reads and writes to variables. For instance, when a function reads a variable, the probable error might propagate from the variable to the function (an edge exist from the variable to the function in the graph). However, when a function writes to a variable, the error propagates from the function to the variable (an edge exist from the function to the variable in the graph). In order to predict the impact of software changes, they navigate graphs from the source of change to the reachable nodes. The authors created 17,000 mutants to investigate how the error that they initiate propagates. Their results show that one of the groups results in better precision and recall and good execution times.

Zhang et al. use function call graphs for similarity detection in file level, which is useful for plagiarism detection, code reuse, and information retrieval methods. They use static analysis methods to construct function call graphs of files. For each node in the graph (each function), they extract structural and functional features and use them to generate the largest common call graphs of two files which will be used to judge the similarity of two files [29].

The similarity between our work and the works in this category is that they try to detect patterns, bugs, and issues. The difference is that we focus on the issues and patterns related to parallel paths.

Chapter 3

Methodology

In this section, we describe the methodology used for generating call graphs and transforming them into an appropriate format that helps us answer research questions. In order to be able to record all of the changes of the project call graph, we need to generate a call graph for each snapshot (the version of the source code after each commit). We first build each version of the source code to get the Jar executable and then use that as the input of java-callgraph tool to get the call graph. Then we run our Python scripts to transform the call graph to our desired format. This approach works efficiently for large projects, but the problem occurs when building the project (especially older versions) is difficult or impossible due to obsolete, or removed dependencies. In contrast, using tools which directly process source code and generate the call graph (without building the project) solves the problems with dependencies, but spends much more time for large projects.

Figure 3.1 shows the whole process from the project source code to generated call graph and specifies the tools and technologies used for each step. We describe the blocks of this figure in the following subsections.

3.1 Corpus and Data

The first block of Figure 3.1 (from the top) is regarding the project source code, so here we describe our dataset. We selected three open-source java projects from Apache repository. We wanted our sample to include large projects with a rather long commit history. Hence, we looked for projects that had at least 10,000 commits and at least five years of commit history. In addition, since we needed exact information about the reason the commit was

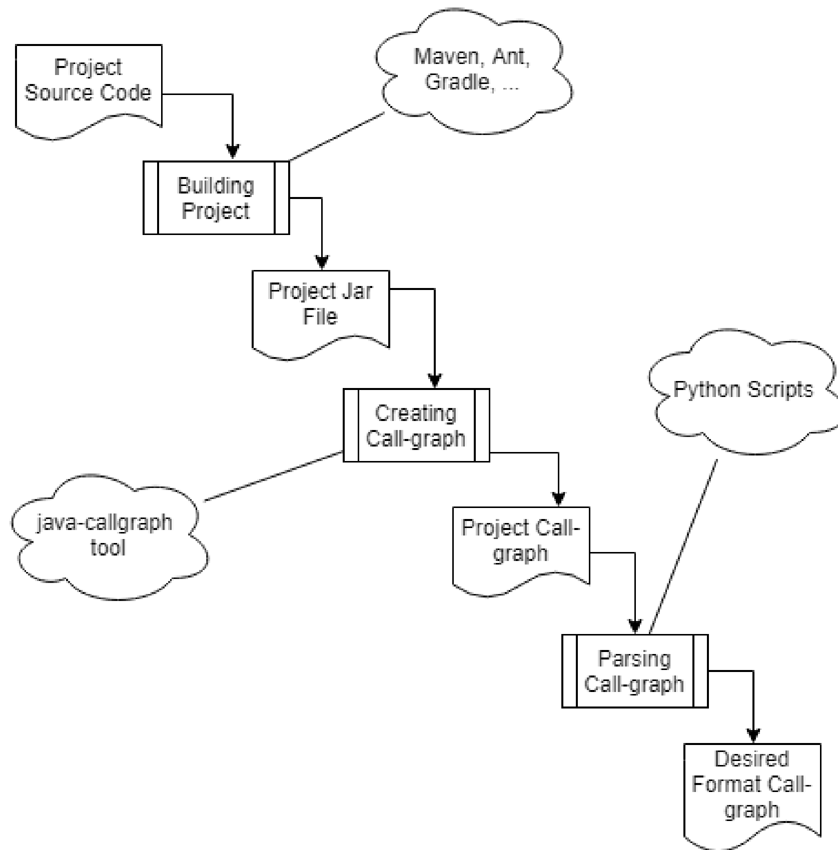


Figure 3.1: The process of generating call graphs

created and the problem it is solving, we looked for projects that have an issue tracking system. Table 3.1 introduces the 3 projects.

Project	Description	# Commits	History
Hadoop	Large Data sets Distributed Processing	21683	10 years
Flink	Stream processing framework	16292	8 years
Jmeter	Load testing tool	16127	20 years

Table 3.1: Dataset projects

Since some of these projects are written in more than one language, we only analyze those commits that include at least one changed Java file.

3.2 Building Projects

The second and third blocks of Figure 3.1 are related to building the project. Since the java-callgraph tool that we use for generating call graphs needs a Jar executable as its input, we had to build each project first. We used the build automation tool that was mentioned in the project documents (e.g., Maven or Gradle). The only issue of this phase is that as we go through a project's commit history, building the older versions of that projects will be more difficult because of removed or obsolete dependencies.

3.3 Generating Call Graphs

The fourth and fifth blocks of Figure 3.1 are related to generating call graphs. There are many approaches and tools for generating call graphs. Some of them work for multiple languages [1] while others only work for a specific language [2][6]. We used java-callgraph [2], which works only for Java applications. This tool gets the built Jar executable of the source code as input and generates its call graph in a specific text format. It also specifies the type of each call (each edge in the call graph) based on Java Virtual Machine specifications [3]. For example, it annotates *Invoke Special* edges, which are special handling for superclass, private, and instance initialization method invocations, with (O).

Figure 3.2a shows a sample Java code snippet and Figure 3.2b shows a graphical representation of the generated call graph for this code sample. Nodes are shown using circles and function calls are denoted using arrows. Also, each edge is annotated with its type. For example, since the edge between functions B.foo() and A.foo() represents a call from subclass to a superclass, it is annotated with (O) which stands for *Invoke Special*. (S) and (M) annotation are related to *Invoke Static* and *Invoke Virtual* respectively.

We use the info from edges for making different queries. For instance, when we are looking for an inheritance relationship between the classes of two methods, we look for edges with type *Invoke Special*.

Every change in the source code might change the call-graph of the project if the change is related to function calls. Figure 3.3 shows an example of a change that changes the call-graph. In this example, function *foo* is removing the call to function *bar*, and because a function call is removed, it will affect the call graph of the code.

Figure 3.4 shows an example of a change that does not affect the call graph. Function *bar* is changing the value of variable $\boldsymbol{a}$ to 5. Since no change is applied to function calls, this change will not affect the call graph of the code.

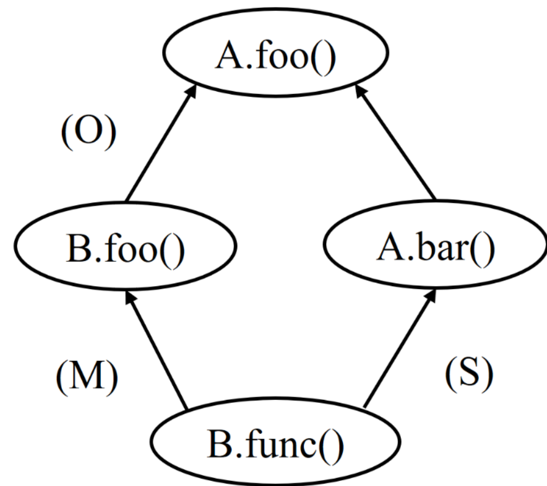
```
class A {
    void foo() {
        // some code here
    }

    static void bar() {
        // some code here
    }
}

class B extends A {
    void foo() {
        super.foo();
        // some code here
    }

    void func() {
        A.bar();
        foo();
    }
}
```

(a) Code snippet



(b) Call graph

Figure 3.2: A sample code snippet and its corresponding call graph

```
class Example {
    void foo() {
        bar();
    }

    void bar() {
        int a = 10;
        System.out.println(a);
    }
}
```

(a) Before change

```
class Example {
    void foo() {
        int b = 10;
        System.out.println(b);
    }

    void bar() {
        int a = 10;
        System.out.println(a);
    }
}
```

(b) After change

Figure 3.3: An example of a change in code which changes the call graph

```
class Example {
    void foo() {
        bar();
    }

    void bar() {
        int a = 10;
        System.out.println(a);
    }
}
```

(a) Before change

```
class Example {
    void foo() {
        bar();
    }

    void bar() {
        int a = 5;
        System.out.println(a);
    }
}
```

(b) After change

Figure 3.4: An example of a change in code which does not change the call graph

3.4 Parsing Call Graphs

The last two blocks of Figure 3.1 are related to parsing call graphs. We use python to parse the output of the java-callgraph tool, which is a text file, and load it in memory as a python map and then serialize it into a text file. In this way, we can load the data quickly whenever we need it.

3.5 Calculating Diffs

When we have the call graph for each version of the project, we calculate the difference between every two successive versions to obtain added and removed functions and added and removed function calls. It enables us to capture the change of project call graph after each snapshot because each function removal results in removing one node and its connected edges and each function call removal leads to removing an edge from the call graph.This process is shown in Figure 3.5. In addition, we capture the commit type for each commit from the project repository. Commit type helps us to determine the type of additions and removals for each commit. For instance, when a commit has Bug tag, its additions and removals are probably related to a bug or a bug fix.

Being able to observe the call graph of the project graphically is useful sometimes. It can help us to understand the structure of the project and its different modules better since it gives us a high-level picture. Also, we need to make different queries to find different patterns in a call graph (e.g., finding all triangular loops in which $\boldsymbol{a}$ calls $\boldsymbol{b}$, $\boldsymbol{b}$ calls $\boldsymbol{c}$, and $\boldsymbol{c}$ calls $\boldsymbol{a}$). We use Neo4j [4], which is a graph database management system to get a high-level picture of the code and to make queries. We also use a Python library named networkx [5] for working with graphs and making queries on them when we do not need a graphical view.

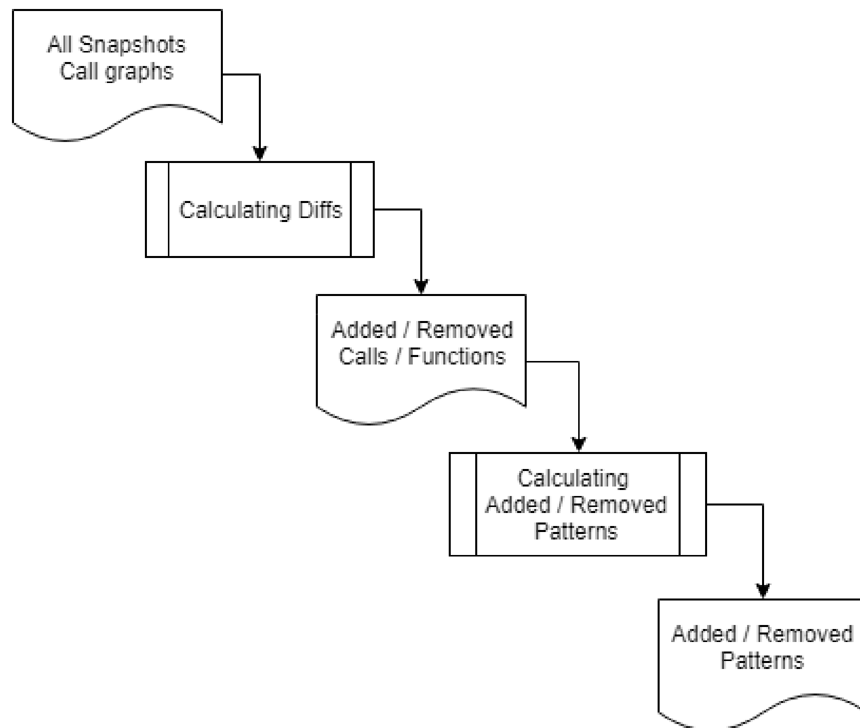


Figure 3.5: The process of finding changed patterns

Chapter 4

Results

In this section, we describe the results of our analyzes regarding the three research questions from chapter 1. In order to simplify the problem, in all of our analyzes we worked on the simplest form of parallel paths which is shown in Figure 4.1 and we call it a triangle from now on. In this case, one function (a) calls two other functions ***b*** and c, and function ***b*** calls function c. Therefore, the question is why *a* needs to call ***c*** in two different ways, directly and indirectly through b.

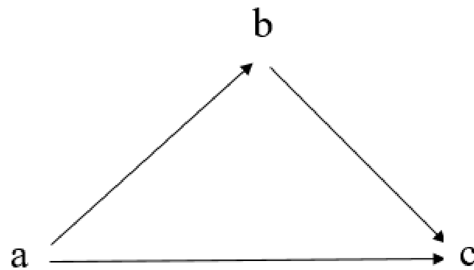


Figure 4.1: The simplest form of parallel paths

4.1 RQ1: Why are parallel paths created and used?

Motivation: This question is important because we need to know the cases that parallel paths are used so as to find the correct and incorrect use of them.

Analysis Approach: In order to answer this question, we first needed to do a manual analysis to find instances of parallel paths in each project, and investigate the reasons they were used. In this regard, we used Neo4j to get all triangles in the form of the triangle in Figure 4.1 for the most recent version of each project. Then we selected 10% percent of these triangles for each project randomly to investigate them manually. After doing this analysis, we were able to categorize triangles in different groups which let us conduct an automated analysis. We created queries for each group and used Neo4j to find all of the instances of that group in each project.

Results: Table 4.1 shows the number of triangles we checked manually and the total number of triangles for each project.

Project	# Triangles	# Analyzed Triangles	Version commit
Hadoop	968	100	252c2b4d52e0dd8984d6f2a8f292f40e1c347fab
Flink	196	20	1e0a77959d27031048c1c4079a504274c82cc173
Jmeter	395	40	f1208484e3a0ac9263d0e43e436ca7ad8fa1749f

Table 4.1: Total number of triangles and analyzed triangles for each project

After this analysis, we were able to build a taxonomy based on what we found and categorize triangles in specific groups. Table 4.2 shows these groups and the number of instances of each group found in manual analysis.

Project	Hadoop	Flink	Jmeter
SubSuperClass	2	1	2
Overloaded Methods	12	3	4
Delegation	10	1	5
Other	76	15	29

Table 4.2: Categorization of triangles

Our investigation revealed that in a large portion of triangles, function ***c*** is a utility function. By utility functions, we mean small functions (usually less than ten lines of

code) that perform a simple task such as getting or setting variables, reading or writing values, checking variable types and similar tasks. When ***c*** is a utility function, it is not that difficult to explain why it is called in two different ways by function ***a***, once directly and once through function ***b*** because many functions need utilities to do simple tasks (e.g. getString or setValue). We categorize such triangles (with ***c*** as a utility function) as a group and call that utility functions. Nevertheless, other groups might have some overlaps with this group. In addition, we found three other categories:

- SubSuperClass: This group represents triangles in which a's class is subclass of b's class and ***a*** calls ***b*** by *super* keyword in Java and both ***a*** and ***b*** call ***c***. Figure 4.2 shows an example of this category.
- Overloaded Methods: In this case, ***a*** and ***b*** are overloaded methods which both call function ***c***, or ***b*** and ***c*** are overloaded methods which ***a*** calls both of them. Figure 4.3 shows an example of overloaded methods category. In order not to confuse names ***a*** and ***b*** with overloaded methods names, we selected different names for functions ***a***, ***b***, and ***c***. In other words, function foo in class A is function ***a***, function bar with one string parameter in class B is function ***b*** and function bar with two parameters in class B is function ***c*** (which is an overloaded version of ***b***).
- Delegation: Delegation category includes triangles in which ***b*** does nothing but calling ***c*** and delegating its work to c. Figure 4.4 shows an example of this category.

After finding these categories by manual analysis, we performed an automated analysis to find instances of each category in the most recent version of each project. To do this, we made queries using Neo4j for each category to find its instances. Here we describe how we make the query for each category:

- SubSuperClass: In this case, ***b*** is a function in class B which is the superclass of class A which contains function ***a***. So a SubSuperClass triangle is a regular triangle in which the type of edge ab is Invoke Special (special handling for superclass, private, and instance initialization method invocations) and the name of class A is not the same as the name of class B.
- Overloaded Methods: In this case, abc is a regular triangle in which functions ***a*** and ***b*** or functions ***b*** and ***c*** reside in a same class and have same names (but different parameters), so the type of ***ab*** or ***bc*** should be Invoke Special.

```
class A extends B {
    void a(){
        super.b();
        C.c();
        // some code here
    }
}
class B {
    void b(){
        C.c();
        // some code here
    }
}

class C {
    static void c(){
        //some code here
    }
}
```

Figure 4.2: Sample code showing SubSuperClass category

- Delegation: In this case, ***b*** is delegating its work to c, i.e., does not do anything but calling c. So ***b*** does not have any outgoing edge in the call graph and resides in a class different from c's class (if the class is same the triangle will be in Overloaded Methods group).

Table 4.3 shows the results of the automated analysis for each category and each project. The results reveal that our detected categories comprise around 13% of triangles in Hadoop, 15% of triangles in Flink, and 9% of triangles in Jmeter. So many of the triangles still reside in utility functions group. We named the fourth category *Other* because, in addition to utility functions, there might be some other categories that we could not find in the manual analysis which "Other" category includes them as well. Nevertheless, after removing all of the instances of three categories from all triangles, we selected a sample of 50 triangles to see whether we can find other categories or not. We found that all of them were different forms or subcategories of utility functions.

```
class A {
    void foo(){
        B.bar("fooString");
        // some code here
        B.bar("barString", 0)
    }
}

class B{
    static void bar(String str){
        bar(str, 0);
    }
    private static void bar(String str, int i){
        // some code here
    }
}
```

Figure 4.3: Sample code showing Overloaded Methods category

Project	Hadoop	Flink	Jmeter
SubSuperClass	11	10	7
Overloaded Methods	65	16	24
Delegation	49	4	6
Other	843	166	358

Table 4.3: Results of automated analysis for each category

```
class A {
    void a() {
        B.b();
        // some code here
        C.c();
    }
}

class B {
    static void b() {
        C.c();
    }
}

class C {
    static void c(){
        // some code here
    }
}
```

Figure 4.4: Sample code showing Delegation Category

4.2 RQ2: Why are parallel paths removed or changed?

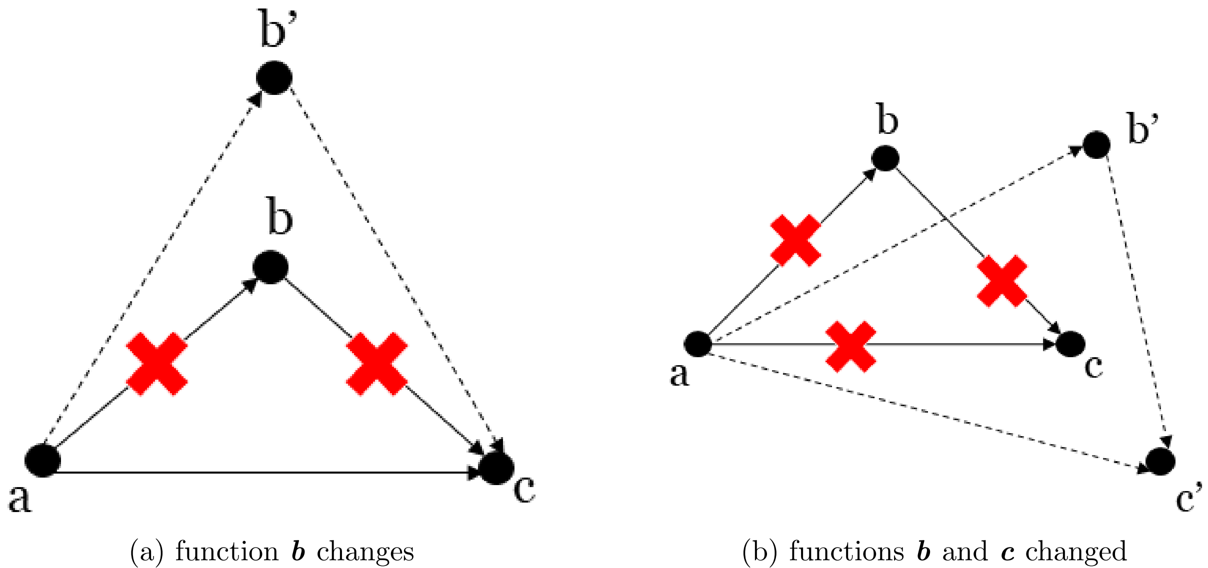
Motivation: This question is crucial for finding patterns related to bugs and issues. Finding the answer of this question helps us to understand the cases that parallel paths are not used in a correct way and should be removed or changed.

Analysis Approach: Parallel paths might be removed or changed because of removals of edges (function calls) or nodes (functions). In each commit, a set of nodes or edges might be removed. We need to check whether each removed node or edge was a part of a triangle or not so as to determine the triangle is removed or not. Since we are curious about paths we focused only on the removal of edges not nodes. So we created some python scripts to go through all of the removed edges through all of the version of the software and find those which were a part of a triangle. Then we categorized all of the removed triangles by their removed edge(s). For example, we put all of the triangles that are removed because of the removal of edge ac in one category.

Results: First, we discuss cases where parallel paths are changed. In many cases, the original triangle (abc) is removed; however, it is not removal but it is a change. For instance, it might be the case that a node is replaced by another node. An example is when function ***b*** in the triangle is an inefficient function and is replaced by b' which is more efficient than b. In this case, edges ab and bc are removed, but edges ab' and b'c are added. This example is shown in Figure 4.5a. Dashed lines show added edges. Figure 4.5b also shows another example of a change in triangles. Another common case of changes in triangles is when refactoring tasks like extract method happen. In Extract Method, a part of a complex function is extracted and moved to a new function which will be called by the old function in order to reduce the complexity of the original function and increase its readability. In terms of call graphs, when an Extract Method happens, one path is extended, i.e., one new node and one new edge will be added to the path. Figure 4.6 shows an example of Extract Method. Dashed lines show added edges.

Here we analyze the cases where parallel paths are removed. Based on the form of parallel paths that we have analyzed (triangles), these paths could be removed or changed because of the removal of any of the nodes ***a***, ***b***, ***c*** or edges ***ab***, ***ac***, ***bc***. However, only two of all of these cases are interesting to us since they have more probability of being related to the problems of parallel paths. When only edge ac is removed, it can be interpreted as a problem in the direct call of ***c*** from ***a*** while indirect call through ***b*** does not have that problem. The situation is similar when edges ab and bc are removed, but edge ac is kept since it can be related to a problem related to indirect call of c.

Using Python scripts, we started from the first commit and analyzed each removed



(a) function ***b*** changes

(b) functions ***b*** and ***c*** changed

Figure 4.5: Parallel Paths Changes

edge to find whether the edge was part of a triangle. In that case, we checked which edge or edges of the triangle are being removed (e.g., edge ab). In order to do this, we had to analyze two versions of the code, the version before the commit and the version after the commit. For instance, to find whether the removed edge was edge ac or not, we checked the version after the commit and looked for paths with length 2 (including two edges) between the source and the destination node of the removed edge. If such a path existed, it meant that the removed edge was ac. Here we present and discuss the results for each project.

4.2.1 Hadoop

Table 4.4a shows the result of this analysis for Hadoop project. The frequency column in this table indicates how many triangles are removed because of the removal of edge(s) in the first column. As we mentioned earlier, a significant amount of removed triangles are related to changes (e.g., refactoring tasks), so in order to separate these cases from real removals, we filtered instances of each category to remove cases related to changes. Table 4.4a shows the results before filtering the data and Table 4.4b shows the results after that. As Table 4.4b shows, categories ab-bc and ab-bc-ac did not have any instance not related to changes.

Since we intended to analyze removals of ac and ab-bc as justified earlier, we only had

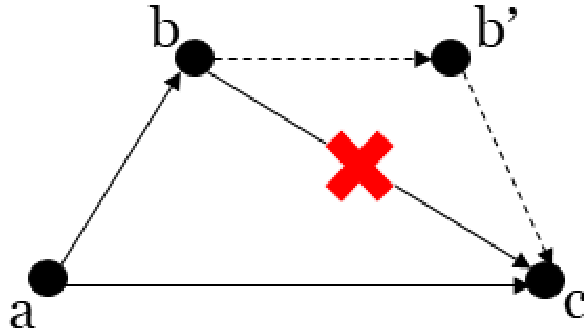


Figure 4.6: Extract Method

Removed Edge(s)	Frequency
ab	16
bc	18
ac	12
ab - bc	3
ab - ac	15
ac - bc	89
ab - bc - ac	7
Total	160

(a) Including Changes

Removed Edge(s)	Frequency
ab	6
bc	11
ac	9
ab - bc	0
ab - ac	4
ac - bc	4
ab - bc - ac	0
Total	34

(b) Without Changes

Table 4.4: Hadoop Removed Triangles

nine instances of ac removal to check. Table 4.5 shows the category of each removal and the frequency of each.

We present the results of the analysis for each project and describe each category of triangles removal in details. For all projects, the "Other" category includes removed triangles that we could not understand the reason of the removal, due either to the complexity of the code or the lack of proper documentation and comments.

- Double Check: In this case, function ***c*** is doing a check (type, existence, not null, and so on) and is called by ***a*** and ***b*** which means both of them are checking the condition. But one of these checks is unnecessary and could be removed. This change could be very helpful specially when ***c*** is an expensive function. Figure 4.7 shows a simple example of this category.

Category	Frequency
Double Check	2
Hook Method	1
Relaxing Conditions	1
Inconsistent Change	2
Logging Issues	1
Parameters	1
Other	1

Table 4.5: Hadoop Removed Triangles Categories

- Hook Method: A hook is a method that is declared in the abstract class, but only given an empty or default implementation and gives the subclasses the ability to implement it and change an algorithm based on their needs. In this case, ***c*** is a hook method and ***a*** and ***b*** are calling it. But these are excessive calls to ***c*** since subclasses might implement ***c*** in a wrong way and mess up the implementation of other functions. So developers decide to remove one of the calls to c, for example, the direct one. Figure 4.8 shows an example of this case.

- Relaxing Conditions: In this case, function ***c*** is doing a check to perform another functionality and function ***a*** does not need this check but function ***b*** needs. So ***a*** removes its call to ***c*** in order not to go through the condition. The figure of this group is almost similar to the figure of Double check category except for function ***c*** which is not called twice necessarily.

- Inconsistent changes: In this case, ac is removed because of a problem in ***c*** or ***a*** problem in the way of calling ***c*** and due to this problem ***bc*** needs to be removed as well, but it is left unattended.

- Logging Issues: Logging information could be done on different levels (e.g., error, debug, warn, info). Logging issues are cases which ***c*** is a logging function and is changed or removed only in one of the direct or indirect paths. One common case is when the logging level is changed (e.g., from warn to debug), which is shown in Figure 4.9.

- Parameters: In this case, ac is removed because of a problem with one or more of the arguments of ***c*** so the problem is not with function ***c*** itself. An example is when a getter function is requesting a deprecated item by using it as its parameter. Because

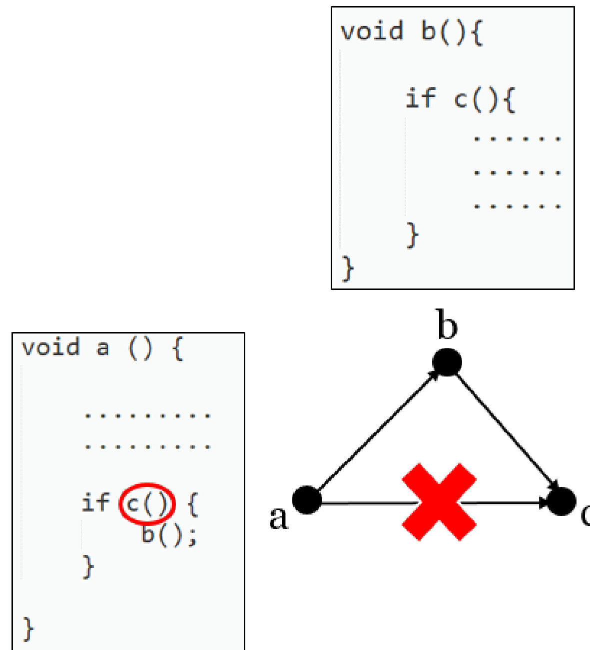


Figure 4.7: Double Check Category

the parameter is deprecated and not used anymore, the call to ***c*** should be removed. Figure 4.10 shows an example of this category.

Based on the results, we see that the instances are scattered, so we have different categories which have one or two instances. Another information that is useful to understand the type of changes and the importance of each category is the type of commit, which includes that change. We extracted the type of each commit from the project repository or its issue tracker. Hadoop and Flink projects use Jira issue tracker while Jmeter uses Bugzilla. Table 4.6 shows commit types related to each category for Hadoop project.

4.2.2 Flink

Table 4.8 shows the result of the analysis for Flink project and Table 4.7 shows the categories. Same as project Hadoop, there were not any instance of ***ab-bc*** category, but only three instances of ac removed triangles. From these three cases, Hook method is already described in Hadoop categories, so we describe SubSuperClass.

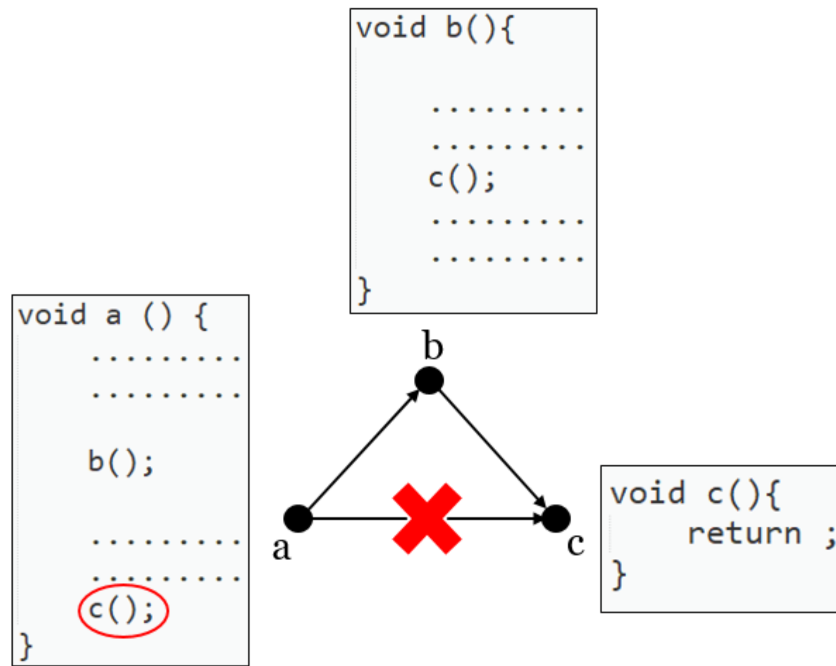


Figure 4.8: Hook Method Category

	Bug	Improvement	Sub-task	None
Double Check		2		
Hook Method				1
Relaxing Conditions		1		
Inconsistent Change	2			
Logging Issues			1	
Parameters		1		
Other		1		

Table 4.6: Commit type distribution between Hadoop categories

- SubSuperClass: In this case, ***abc*** is a triangle of SubSuperClass group and function ***a*** which is in the subclass removes its call to ***c*** but function ***b*** which is in super class keeps it so a triangle of SubSuperClass category is removed because of the removal of the edge ***ac***.

Table 4.9 shows commit types related to each category for Flink project.

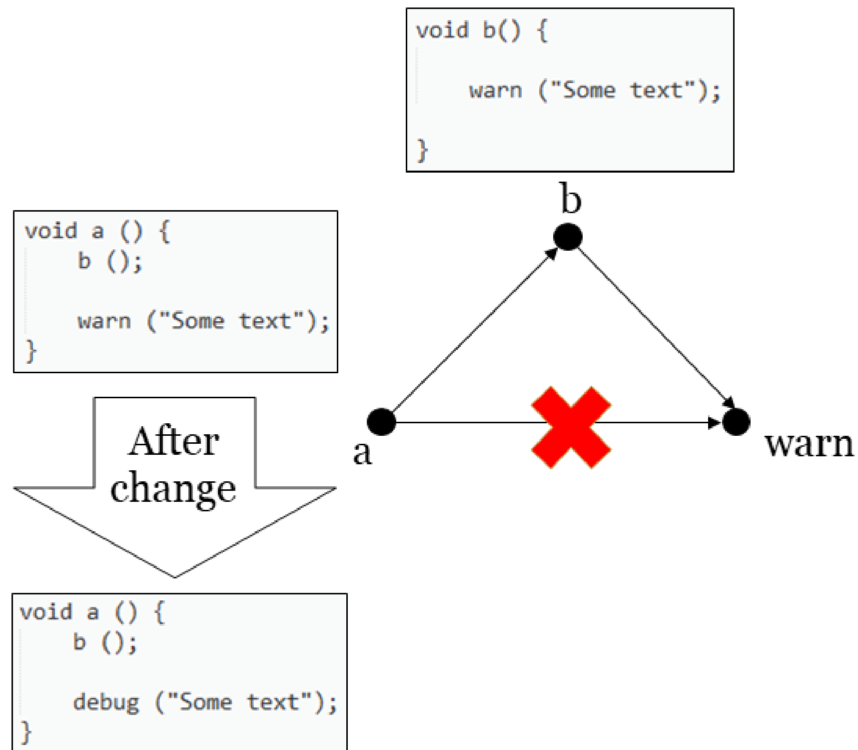


Figure 4.9: Logging Issues Category. In function ***a***, warn is replaced with debug.

Although the instances in Flink project are too few to enable us to make strong conclusions, they are interesting and useful because two of them are introducing SubSuperClass which is a new group (compared to Hadoop categories) and the other one that is in Hook Method group, adds strength and validity to another instance of this category in Hadoop project. In other words, if we find only one sample for a category, it might be the case that it is an accidental mistake and is not a common problem. However, the more instances that we find for each category in different projects, we can conclude more confident that we have found a common issue.

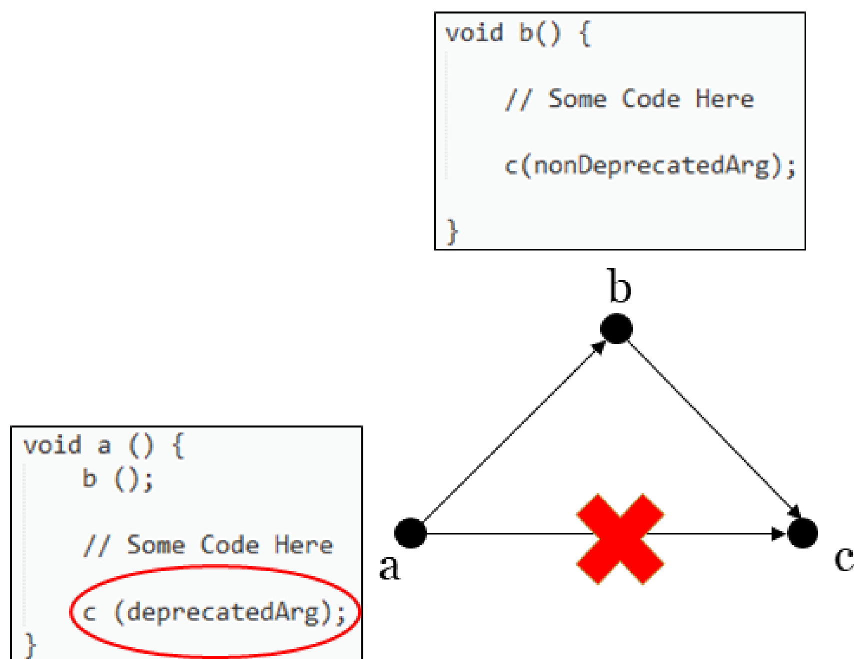


Figure 4.10: Parameters Category. In function ***a***, the call to ***c*** should be removed because it is called with a deprecated parameter.

4.2.3 Jmeter

Table 4.10 shows the result of the analysis for Jmeter project and Table 4.11 shows the categories.

Overloaded: In this case, ***abc*** is a triangle of overloaded methods group in which ***b*** and ***c*** are overloaded methods and ***a*** removes its call to ***c*** to call ***c***', another overloaded version of ***c*** with more parameters but ***b*** keeps the call to ***c***.

Incomplete Change: In this case, the removal of parallel paths (***ac***) is a part of a larger change which includes the removal of the longer path (abc) later on. So ac is removed in commit c1 at time t1 and bc is removed in commit c2 at time t2 (t2 ¿ t1). So we can explain the removal of ac as an incomplete change. One example of this group which exists in all three projects is the change of logging package. When the logging package needs to be changed for the whole project, different classes might add their dependency to the new package and remove it from the old package at different times.

Table 4.12 shows commit types related to each category for Jmeter project.

Removed Edge(s)	Frequency
ab	5
bc	3
ac	3
ab - bc	0
ab - ac	9
ac - bc	35
ab - bc - ac	0
Total	55

(a) Including Changes

Removed Edge(s)	Frequency
ab	3
bc	3
ac	3
ab - bc	0
ab - ac	0
ac - bc	2
ab - bc - ac	0
Total	11

(b) Without Changes

Table 4.7: Flink Removed Triangles

Category	Frequency
SubSuperClass	2
Hook Method	1

Table 4.8: Flink Removed Triangles Categories

	Bug	Improvement	Sub-task	None
SubSuperClass		2		
Hook Method		1		

Table 4.9: Commit type distribution between Flink categories

4.2.4 Conclusions

In these categories, we are curious to know which of them are related to parallel paths problems. In other words, we want to find the cases where double calling a function, once directly and once indirectly causes a bug or a problem (e.g., efficiency) or makes the code more complex and more difficult to understand and change. We describe whether each category is related to parallel paths or not.

- Double Check: The problem in this category is obviously related to parallel paths because a checking function is called twice whereas two calls are not necessary and one of them is enough. Therefore, the solution to this issue is breaking the parallel path and using only one branch of that.

Removed Edge(s)	Frequency
ab	19
bc	40
ac	46
ab - bc	3
ab - ac	18
ac - bc	59
ab - bc - ac	1
Total	186

(a) Including Changes

Removed Edge(s)	Frequency
ab	14
bc	33
ac	20
ab - bc	0
ab - ac	0
ac - bc	2
ab - bc - ac	0
Total	69

(b) Without Changes

Table 4.10: Jmeter Removed Triangles

Category	Frequency
Overloaded	1
Inconsistent Change	3
Parameters	3
Incomplete Change	8
Other	5

Table 4.11: Jmeter Removed Triangles Categories

	Bug	Improvement	Sub-task	None
Overloaded		1		
Inconsistent Change	2	1		
Parameters	3			
Incomplete Change	8			
Other	4	1		

Table 4.12: Commit type distribution between Jmeter categories

- Hook Method: In this category, function ***a*** removes its call to the hook method because it is calling hook method through function ***b*** and another direct call will be useless or even harmful when sub classes implement the hook method in a bad or wrong way. So we can state that in this category the problem is with double calling a function and only one call to that function is enough.

- Relaxing Conditions: In this case, function ***c*** is doing a check to perform a functionality, but ***a*** does not this check and wants to perform the functionality directly so removes its call to ***c***, but ***b*** still needs the check. So the change is related to functions ***a*** and ***b*** and not related to double calling ***c***.

- Inconsistent Change: In this category, the problem is not related to parallel paths definitely, because we believe that the indirect path which is not removed should be removed as well, but is left due to a mistake. This category helps to find bugs in the source code which will be discussed in section 4.4 in details.

- Logging issues: In this category, the problem is related to inappropriate use of a logging function in function ***a*** which results in the removal of edge ac. So the problem is in function ***a*** and is not related to the fact that ***c*** is being called by ***a*** and ***b***.

- Parameters: In this case, the problem is not with function c, but with its parameters which might be obsolete or deprecated. Therefore, the problem is not related to double calling a function.

- SubSuperClass: In this case, function ***a*** which is in a subclass is removing its call to function ***c*** but function ***b*** which is in super class keeps the call. So we can conclude that there is no need for double calling function ***c*** in both ***a*** and ***b*** and one of them should be removed.

- Overloaded: In this case, function ***a*** needs to call an overloaded version of ***c*** which has more parameters to provide more functionality, but function ***b*** does not need that. So the reason of change is not that ***a*** and ***b*** both call ***c***.

- Incomplete change: In this category, the removal of ***ac*** is a part of a larger change which will remove the indirect path as well. Hence, the problem is not related to parallel paths and double calling c.

To sum up, from above categories, the problem in Double Check, Hook Method, and SubSuperClass is related to parallel paths and the fact that both ***a*** and ***b*** call ***c***. In other cases, the problem is related to either ***a*** or ***c*** or their parameters.

In addition, commit type distribution tables show that inconsistent change category is almost always associated with bugs. Also, other categories related to parallel paths such as Double Check, Hook Method, and SubSuperClass are usually related to improvement commits and not bug commits.

4.3 Comparison of the first and the last versions

Regarding research question 1, we categorized triangles based on the structure of them, and we found three main categories, SubSuperClass, Overloaded Methods, and Delegation. In this section, we did an analysis to compare the number of each category in the first and last version of each project. It enabled us to find the reasons each category was removed. For example, we wanted to understand why triangles in SubSuperClass group might be removed and whether there is a correlation between a specific type of removed and a category. In order to do this, we used Neo4j to run the query of each category on the first and the last version of each project. Then we compared the results for the first and the last version to find added, remained, and removed instances. We explain and analyze the results for each project.

4.3.1 Hadoop

Table 4.13 shows the results of comparison for Hadoop project.

	First Version	Last Version	Remained	Added	Removed
SubSuperClass	11	15	9	6	2
Overloaded	35	65	31	34	4
Delegation	34	49	32	17	2

Table 4.13: Hadoop first and last versions comparison

Based on the results, 8 triangles in three groups were removed. We analyzed these removals and put them in similar groups. Table 4.14 shows this categorization with the frequency of each category.

Here we describe each category in details:

- Logging Package Change: The project changes its logging package, and every edge that was connected to one of the logging functions from the old logging package is removed.
- Deprecated Package Removal: An obsolete useless package is removed, and all of the triangles including the functions of that package are removed.

	SubSuperClass	Overloaded	Delegation
Logging Package Change	1	0	0
Deprecated Package Removal	1	0	0
Moving Classes Between Packages	0	2	0
Changing Overloaded Methods	0	1	0
Breaking Overloaded	0	1	0
Deprecated Parameters	0	0	1
Changing Mechanism	0	0	1

Table 4.14: Hadoop removed triangles of first version categories

- Moving Classes Between Packages: one new package is created which aggregates all of the contents of two previous packages.
- Changing Overloaded Methods: Functions ***a*** and ***b*** which are parser functions are changing the mechanism of stream reading, so remove the reader from their parameters and use a factory which is a global variable instead.
- Breaking Overloaded: Functions ***a*** and ***b*** are overloaded methods where ***b*** handles specific types of input. Developers decide that ***b*** should not use ***c*** due to efficiency issues, but ***b*** should define an alternative that does the functionality of ***c*** more efficiently. So they rename ***b*** (because it is not doing something similar to a anymore) and remove the call to ***c*** from it.
- Deprecated Parameters: One or more deprecated arguments of a function should be removed, so the function is removed because there is no need for it anymore.
- Changing serialization mechanism: New serialization mechanism is used, and the old one is removed because it had backward-compatibility issues. So all of the functions related to serialization (e.g., read and write) use the new mechanism and remove their calls to methods related to the old serialization approach.

4.3.2 Flink

Table 4.15 shows the results of comparison and Table 4.16 shows the categorization or removals with the frequency of each category for Flink project.

	First Version	Last Version	Remained	Added	Removed
SubSuperClass	11	10	4	6	7
Overloaded	24	16	16	0	8
Delegation	3	4	2	2	1

Table 4.15: Flink first and last versions comparison

	SubSuperClass	Overloaded	Delegation
Consolidating Related Features	2	0	0
Dependencies Changes	5	8	0
Hook Methods	0	0	1

Table 4.16: Flink removed triangles of first version categories

Here we describe each category in details:

- Consolidating related features: all of the classes containing utility functions related to serialization are consolidated and merged into a single utility class.
- Dependencies Changes: There are some dependencies to packages out of Flink project, and many sub-modules in Flink use these packages. It causes some problems. For example, the classes of these packages appear in the classpath multiple times which is unclean. In addition, some of the dependencies require to include license files, and it is difficult to build a good automatic solution for that. Thus, they build and deploy a version of the external packages inside the project which solves the mentioned problems.
- Hook Methods: Function $\boldsymbol{b}$, which is a hook method is refactored in order to make the interface more restricted to avoid users mess up the implementation. So function $\boldsymbol{a}$ is changed and removed its call to function $\boldsymbol{c}$ because it can get its data from $\boldsymbol{b}$ as a result of refactoring.

4.3.3 Jmeter

Table 4.17 shows the results of comparison and Table 4.18 shows the categorization or removals with the frequency of each category for Jmeter project. Since Logging Package Change is explained in Hadoop categories, we describe the other three categories here.

	First Version	Last Version	Remained	Added	Removed
SubSuper	8	7	7	0	1
Overloaded	24	24	18	6	6
Delegation	6	6	3	3	3

Table 4.17: Jmeter first and last versions comparison

	SubSuperClass	Overloaded	Delegation
Logging Package Change	1	0	2
Changing Access	0	0	1
Removing Useless Functions and Parameters	0	2	0
Overloaded to Extend Functionality	0	4	0

Table 4.18: Jmeter removed triangles of first version categories

Here we describe each category in details:

- Changing Access: Function ***a*** is renamed and is changed to private from public. A new public function with the previous name of ***a*** is created. This new function calls previous function ***a*** and adds some functionality to it. Therefore, users should not be able to call the old ***a*** directly, but they should call it indirectly from new a, so we can state that the access of users to function ***a*** is changed and limited.

- Removing Useless Functions and Parameters: Sometimes functions are removed because their useless parameters should be removed. For example, a call to a function which creates menu items might be removed because a menu item is not needed.

- Overloaded to Extend Functionality: One function needs more parameters to implement new functionality, so an overloaded version of that function is used. For instance, a function which creates menu items needs one more parameter to define mnemonics for menu items.

4.3.4 Conclusions

Based on the results of this section we can conclude that:

- For all three projects, most cases of SubSuperClass group are related to removing or changing dependencies to some packages.
- In Hadoop and Jmeter projects, there is a similar pattern for overloaded methods. In Hadoop, we have the category *Changing Overloaded Methods* in which one parameter is removed from overloaded methods, and in Jmeter we have overloaded to extend functionality which adds one more parameter to previous overloaded functions. Therefore, both are associated with overloaded methods changes.
- In Flink and Jmeter projects, one common pattern for Delegation category is that for the Hook Methods category in Flink and Changing Access category in Jmeter there is some kind of change in the interface which prevents or limits users to mess up the implementation.
- In some cases (e.g., Consolidating Related Features in Flink or Changing Overloaded Methods in Hadoop), the problem is related to the category. For example, for consolidating related features in Flink, some classes that were derived from a super class were removed and merged into a single class. So we can interpret it in this way that some useless SubSuperClass triangles were removed, so the problem is related to misuse of SubSuperClass category. Also, in Changing Overloaded Methods, the problem is using an incorrect overloaded version of a function which should be replaced by a correct one.

4.4 RQ3: Is it possible to suggest to developers to apply some changes to the current version of the software based on the information gained from analyzed parallel paths?

Motivation: The importance of this question is that if we can find some ways to find the cases of the patterns that we have found in the previous versions of each project, it will enable us to detect problems in the current version of projects.

Analysis Approach: Suggesting changes to developers should be based on their changes applied to previous versions of the software. We analyzed removed and changed triangles for answering RQ2 and categorized removals and changes. Then we created queries for some of the categories to find the instances of them in the most recent version of each project. Then we needed to remove false positives and keep the real cases of each category. It enabled us to make suggestions for either removing some bugs and mistakes from code or refactoring it and making it more readable and understandable.

Results: Finding instances of each category in the last version of each project is not an easy task since we have limited information about each category and a few instances of that. Nevertheless, for some of the categories, it is possible to make queries to find instances of that category and then try to find those cases which could be changed or improved. We only focused on categories that are related to parallel paths issues. Here we describe these groups and elaborate on the approach that we used.

Double Check: In this group, $\boldsymbol{c}$ is a function which is doing a check and is called by $\boldsymbol{a}$ and $\boldsymbol{b}$. In order to find instances of this group, one approach is to look for triangles in which the name of function $\boldsymbol{c}$ starts with a word which is related to checking (e.g. is, has, check, contains). We used this query and then removed false positives, i.e., cases in which the word that we are looking for is a part of another word. For example, *is* could be a part of *visit*. Table 4.19 shows the results of this analysis. It reveals that there are 6 cases of double check in Hadoop project, 4 cases in Flink project, and one case in Jmeter project which could be changed in order to either remove some unnecessary code or improve efficiency by removing checks. Correct cases in the table are those triangles in which the edge ac could be removed since function $\boldsymbol{b}$ is able to perform the check for $\boldsymbol{a}$ and run the correct functionality based one that. In addition, in some correct cases, $\boldsymbol{bc}$ could be removed because no function in the source code calls $\boldsymbol{b}$ without checking $\boldsymbol{c}$ before that, so $\boldsymbol{b}$ is doing an unnecessary check and can remove that. Indeed we checked each instance that our query found to check whether it is a correct instance of double check category or

not. For example, for the Hadoop project, in the 22 cases that were not correct, it is not possible to remove the call to ***c*** from neither ***a*** nor ***b***.

Project	Query Result	Filtered Result	Correct Cases
Hadoop	105	28	6
Flink	33	18	4
Jmeter	23	9	1

Table 4.19: Double Check Group Instances

Hook Methods: In this category, all ***a*** and ***b*** and ***c*** reside in a same class and ***a*** and ***b*** call ***c*** which is a hook method and either is a native function (without implementation) or does a very simple task and lets sub classes to override it and change the implementation based one their needs. So we have two criteria that could be used for making queries for hook methods. In other words, we need all of the triangles which ***a***, ***b***, and ***c*** are in the same class and ***c*** is not calling any other function. We executed this query for each project and Table 4.20 shows the results. Based on the results, we only could find two cases of hook methods in Hadoop project having the problem of double calling function ***c***.

Project	Query Result	Filtered Result	Correct Cases
Hadoop	40	14	2
Flink	4	2	0
Jmeter	9	0	0

Table 4.20: Hook Methods Group Instances

SubSuperClass: Regarding this category, we only found two instances in Flink project, and these two cases were related to a refactoring task that was very specific to Flink project and can not be generalized. So there is no obvious way to find other instances of this category in each of the three projects.

Inconsistent Changes: Although we categorized this group as groups that are not related to parallel paths problems, we believe it is essential to focus on this group since it can reveal developers' mistakes in applying inconsistent changes. In this category, one branch of the parallel paths is removed, and the other branch is left, but that is a mistake, and both should be removed. Finding instances of this category is only possible by analyzing all cases of removal of edge ac and find those that are removed inconsistently. We found

two instances of this group in Hadoop (22% of ac removed triangles) and three instances in Jmeter project (15% of ac removed triangles).

The two cases in Hadoop are related to avoiding logging some secure information. So all of the logging functions that were using that information as their arguments were removed. However, in one case, developers forgot to remove the logging function. We could find this by analyzing the removal of an ac edge while ab and bc were kept. So developers should have removed bc as well.

In two cases in Jmeter, $\boldsymbol{c}$ is a logging function called by $\boldsymbol{a}$ and concatenates some variables and uses the result as one parameter. In one commit, developers have separated these variables and called an overloaded version of logging function which accepts those as its parameters. Therefore, concatenation which calls toString function for each variable and might be inefficient sometimes, will not happen anymore. In the triangle, only function $\boldsymbol{a}$ is changed, but function $\boldsymbol{b}$ also has the same condition and uses concatenation, so should be refactored but is left unattended.

Regarding the other case in Jmeter, $\boldsymbol{c}$ is a logging function and function $\boldsymbol{a}$ removes its call to $\boldsymbol{c}$ because there is a throw statement which is enough due to developers decision, so they remove the call to $\boldsymbol{c}$ because they think it is useless and confusing to users. Based on this decision they should have refactored function $\boldsymbol{b}$ and removed $\boldsymbol{c}$ from that as well but it is left in the code.

Almost all of these cases are related to bugs, i.e., the edge ac was removed because of a problem in $\boldsymbol{c}$, which shows the importance of focusing on this group.

Chapter 5

Conclusions

In conclusion, we have proposed an approach to analyzing parallel paths in function call graphs. Using three open-source large Java projects as our data set, we answered three research questions. We used the simplest form of parallel paths which is a triangle for the sake of simplicity. First, we concluded that in triangles, the function that is called by the other two functions (function c) is usually a utility function which performs very simple tasks. In addition, we found some other groups and categorized triangles in these groups. Based on the structure of each group and the information we gained from our analysis, we were able to create queries to find instances of each category in the projects. Regarding the removals and changes of parallel paths, we stated that parallel paths are mainly changed during refactoring tasks like extract method, which is the most common. Also, after categorizing triangles removals, we were able to separate those categories which are associated with parallel paths and the problem in them is double calling a function by two other functions in the triangle. Finally, based on the cases and categories that we found in our analyses, we made queries to find instances of some of the categories in the most recent version of each project. It enables us to suggest some changes to developers to avoid bugs and problems or reduce the complexity and improve the readability of it. We believe that using this approach for more complex structures enables us to make more accurate queries and find instances of each category easier. That is because, in contrast to triangles that are the simplest form of parallel paths, complex structures give us more detailed information that could be used for filtering and making queries.

5.1 Threats to Validity

There are some threats to our results and conclusions. First, we only used three projects as our data sample. Although these projects are large well-known projects, all of them are open-source projects from Apache repository, so they might not be a good representative for all software projects. Also, all of the projects are in Java, so the analysis of other languages might result in different conclusions. Moreover, because our tool for graph generation needed Jar file of the project, we were not able to perform our analyses for the whole history of each project due to the problems of building earlier versions (e.g., dependency on obsolete packages). Furthermore, the categorization of changes and removals is based on our understanding of the source code using commit messages, comments, and issue tracking system for each project. Therefore, there might be some misunderstandings or mistakes in our comprehension of source code. Finally, in our suggestions to developers, we do not consider optimizations that compilers might apply. For example, when we suggest developers to remove a double check to improve the performance, some compilers might consider this and bypass the double check. However, we assume that compilers do not perform such optimization tasks.

5.2 Future Work

There are different paths for continuing this work. First, one can use more complex forms of parallel paths for analyses. We used only the simplest form of parallel paths (triangles) for simplicity. There might be more interesting results in other forms of parallel paths like diamonds or structures with more than two parallel paths. Moreover, more complicated paths are easier to detect because they provide us with more information about the structure of parallel paths and help us to create more precise queries for detection. In addition, analyzing more projects will result in finding new categories and creating new queries to detect more instances of candidates for change or removal.

References

[1] Doxygen. http://www.doxygen.nl/, 2018.

[2] java-callgraph. https://github.com/gousiosg/java-callgraph, 2018.

[3] Java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html.

[4] Neo4j. https://neo4j.com/.

[5] Networkx. https://networkx.github.io/, 2014.

[6] Python call graph. http://pycallgraph.slowchop.com/en/master/, 2018.

[7] Subdue sgiso algorithm. https://github.com/gromgull/subdue/blob/master/src/sgiso.c, 2012.

[8] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429, June 2012.

[9] Johannes Bohnet and Jürgen Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06, pages 95–104, New York, NY, USA, 2006. ACM.

[10] Y. Du, J. Wang, and Q. Li. An android malware detection approach using community structures of weighted function call graphs. *IEEE Access*, 5:17478–17486, 2017.

[11] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54, New York, NY, USA, 2013. ACM.

[12] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. Code2graph: Automatic generation of static call graphs for python source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 880–883, New York, NY, USA, 2018. ACM.

[13] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, October 1997.

[14] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai. Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1109–1113, May 2017.

[15] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, pages 101–104, New York, NY, USA, 2018. ACM.

[16] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4):233–245, Nov 2011.

[17] M. Madhan, I. Dhivakar, T. Anbuarasan, and C. Thirumalai. Analyzing complexity nature inspired optimization algorithms using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1077–1081, May 2017.

[18] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998.

[19] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A large-scale study of call graph-based impact prediction using mutation testing. *Software Quality Journal*, 25(3):921–950, Sep 2017.

[20] M. Oruc, F. Akal, and H. Sever. Detecting design patterns in object-oriented design models by using a graph mining approach. In *2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 115–121, April 2016.

[21] D. Qingfeng, S. Kun, Y. Kanglin, and Q. Juan. Metrics analysis based on call graph of class methods. In *2017 International Conference on Progress in Informatics and Computing (PIC)*, pages 18–24, Dec 2017.

[22] Yu Qu, Xiaohong Guan, Qinghua Zheng, Ting Liu, Lidan Wang, Yuqiao Hou, and Zijiang Yang. Exploring community structure of software call graph and its applications in class cohesion measurement. *Journal of Systems and Software*, 108:193 – 210, 2015.

[23] Simone Romano, Giuseppe Scanniello, Carlo Sartiani, and Michele Risi. A graph-based approach to detect unreachable methods in java software. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1538–1541, New York, NY, USA, 2016. ACM.

[24] P. Sawadpong and E. B. Allen. Software defect prediction using exception handling call graphs: A case study. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 55–62, Jan 2016.

[25] M. D. Shah and S. Z. Guyer. An interactive microarray call-graph visualization. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 86–90, Oct 2016.

[26] Burak Turhan, Gzde koak, and Ayse Bener. Data mining source code for locating software bugs: A case study in telecommunication industry. *Expert Syst. Appl.*, 36:9986–9990, 08 2009.

[27] Wikipedia. Call graph. https://en.wikipedia.org/wiki/Call_graph.

[28] P. Wu, J. Wang, and B. Tian. Software homology detection with software motifs based on function-call graph. *IEEE Access*, 6:19007–19017, 2018.

[29] Jin Zhang, Dahai Jin, and Yunzhan Gong. File similarity determination based on function call graph. In *2018 IEEE International Conference on Electronics and Communication Engineering (ICECE)*, pages 55–59, 12 2018.

[30] S. Zhang, J. Ai, and X. Li. Correlation between the distribution of software bugs and network motifs. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 202–213, Aug 2016.

[31] H. Zhou, W. Zhang, F. Wei, and Y. Chen. Analysis of android malware family characteristic based on isomorphism of sensitive api call graph. In *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pages 319–327, June 2017.